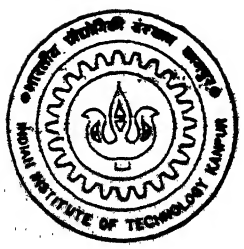
Parallel Algorithms for some Optimization Problems

by

K JEEVAN MADHU



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

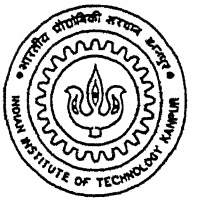
Indian Institute of Technology, Kanpur

MAY, 1998

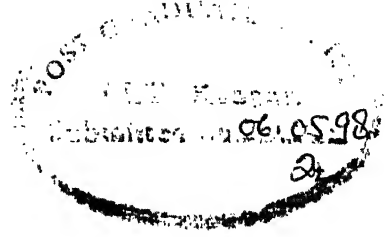
Parallel Algorithms for some Optimization Problems

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by
K Jeevan Madhu



to the
Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
May, 1998



Certificate

Certified that the work contained in the thesis entitled "*Parallel Algorithms for some Optimization Problems*", by *Mr. K Jeevan Madhu*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Sanjeev Saxena)
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

May, 1998

To my beloved Lord Sri Venkateswara

Abstract

We propose sequential and parallel algorithms for three different *vehicle routing* problems. We propose a sequential algorithm and a parallel one for the *all pairs shortest routing* problem. For the *single vehicle routing* problem, when all the locations are on a line, and having either deadline time or release time associated with them, we propose an $O(n^4 \log n)$ cost, and $O(n)$ time optimal, parallel algorithms. We propose a sequential and two parallel algorithms for the *traveling repairman* problem, when all the locations are on a line, and having no time windows associated with them.

For the problem of *operator coalescing with precedence constraints*, we propose an $\mathcal{NC}$ algorithm that runs in $O(\log^2 n)$ time with $O(n^3 \log n)$ cost on a CREW PRAM. The problem here is to schedule operations of two jobs, such that the schedule gives maximum reward we get, when operations are coalesced.

We propose sub-cubic cost algorithms for the *all pairs longest path* (APLP) problem and the *all pairs longest distance* (APLD) problem for directed acyclic graphs. The first parallel algorithm solves the APLD problem, for a directed acyclic graph with unit edge costs. The second one solves the APLP problem and consequently APLD problem for a directed acyclic graph with non negative edge costs.

We present an $\mathcal{NC}$ algorithm for scheduling n unit length tasks, on m identical processors for the case, where the precedence constraint is an *interval order*. The algorithm produces the same schedule as the one produced by the list scheduling algorithm. Here, we reduce cost of best known parallel algorithm by a factor of $O(\log^2 n)$ time.

Acknowledgements

I would like to express my sincere gratitude to Dr Sanjeev Saxena, whose immense
›wledge in the field, with active participation and constant encouragement guided me
/ards the completion of the thesis. It has been great pleasure to work under his able
dance.

I would like to thank my examiners Dr A. Mukhopadhyaya and Dr D. Manjunath for
:ir useful comments on thesis. I would also like to thank Dr Ramesh Krishnamurti for
useful explanations.

Special thanks to Gore, Kataru, Kishore, Gorti, Tatik, Sandeep, Yudhdhar, Vihari,
ıl, Kshit, Mukkam, Rajesh, Dhanabal, Pratima, Samarjit, Himadri, and Murali for
:ir support during the period of thesis.

Thanks to all my Hall IV friends Bal, Banda, Gundu, Chotu, Motu, and Avesam for
:ir memorable company during my stay in IITK.

Jeevan Madhu

Contents

1 Introduction 1

1.1 Problems 2

1.2 Model of Computation 4

1.3 Organization of the Thesis 5

2 Preliminaries 6

2.1 Basic Parallel algorithmic techniques 6

2.2 Basic Graph Definitions 7

2.3 Shortest path algorithm for Layered Directed Acyclic Graph 8

2.4 Fibonacci heaps 9

2.5 Analyzing Algorithms 11

2.5.1 Order Notation 11

2.5.2 Complexity classes 11

3 Parallel Algorithms For Vehicle Routing Problems 13

3.1 Introduction 13

3.2 Preliminaries 17

3.3 All pairs shortest routing problem 18

3.3.1 Sequential algorithm for the APSR problem 19

3.3.2 Parallel Algorithm for the APSR problem 20

3.4 Parallel algorithms for dSVRPTW-line and rSVRPTW-line Problems . . . 28

3.4.1 Sequential algorithm for dSVRPTW-line 28

3.4.2 Parallel algorithms for dSVRPTW-line problem 29

3.5 Traveling Repairman Problem 33

3.5.1 Sequential algorithm 34
3.5.2 Parallel algorithms for TRP-line problem 35

4 Parallel Algorithms For Coalescing Operations with Precedence Constraints In Real-time Systems 39
4.1 Introduction 39
4.2 Scheduling two periodic jobs 40
4.2.1 Sequential algorithm 40
4.2.2 An example 41
4.3 Parallel algorithms for scheduling two jobs 42
4.3.1 Layered graph construction 44
4.3.2 $O(n)$ time Optimal parallel algorithm 45
4.3.3 $O(\log^2 n)$ time parallel algorithm 46

5 Sub-Cubic Cost Algorithm for the All Pairs Longest Path Problem for Directed Acyclic Graph 49
5.1 Introduction 49
5.2 Basic definitions 50
5.3 Longest Distance matrix multiplication by divide-and-conquer 50
5.3.1 Distance matrix multiplication by table-lookup 51
5.3.2 Computation of table I 53
5.3.3 Determination of size of submatrices 53
5.3.4 Parallelization 53
5.4 All pairs longest paths 54
5.4.1 Parallelization for graphs with unit costs 56
5.4.2 Parallelization for graphs with general costs 57

6 Scheduling Interval Ordered Tasks In Parallel 60
6.1 Introduction 60
6.2 Basic Definitions 61
6.3 Sequential Algorithm 62
6.4 Parallel Algorithm 63
6.4.1 Computing optimal schedule length 63

6.4.2 Parallel algorithm for constructing optimal schedule 65

7 Conclusions **69**

Bibliography **71**

List of Tables

1 The complexity of special cases of SVRPTW-line(n is the number of jobs) 15

2 The complexity of special cases of TRPTW-line(n is the number of jobs) 16

3 Reward table of the example 41

List of Figures

1 An example of Layered graph 8

2 An example Graph G with time windows and travel times. Three tables specify the cost vector as well as optimal routes for each row 21

3 An example describing our parallel algorithm for TRP-line problem 38

4 An example of scheduling operations with precedence constraints 42

5 Information flow in equations given in Section 4.2.1 42

6 Information Flow in modified equations 44

Chapter 1

Introduction

In recent years, there has been tremendous effort on the part of researchers to develop fast and efficient parallel algorithms. Advances in VLSI technology have provided a cost-effective means of obtaining increased computational power by way of multiprocessor machines. These machines consist of a few processors to many thousands and potentially millions of processors. Parallel processing methodology involves employing more than one processor to a task and makes them cooperate in various ways to solve computationally intensive problem. However, there is a strict upper bound on the processor speeds. This bound is dictated by the speed of light and the minimum distance required on VLSI chip between two components (so that they do not interact with each other). Hence, the hardware technology alone may not satisfy the very increasing demands of computational power. Although we can parallelize some operations of sequential algorithms (through software), it won't be effective as a number of complex operations pose problems. So, it appears that parallel model of solving problems is the only practical approach to exploit the (massive) parallelism available from these multi processor machines.

An algorithm is a sequence of steps (to solve a particular problem). A sequential algorithm specifies the actions to be taken by a single processor for solving the problem. In a sequential algorithm, only a single instruction is executed at any time. In a parallel algorithm, the algorithm is executed by more than one processor simultaneously thereby reducing the computation time. We have to extract the parallelism inherent in the problem to obtain good parallel algorithms. In this thesis, we have tried to develop some

efficient parallel algorithms for some optimization problems. In Section 1.1, we give a brief introduction to the problems chosen for this thesis. In Section 1.2, we give a brief introduction to the deployed model of computation.

1.1 Problems

In this section, we introduce the problems that have been considered under this thesis. Following problems have been considered.

1. Vehicle routing problems

Consider a complete directed graph in which each arc has a given length. There is a set of jobs, each job i located at some node of the graph, with an associated processing time or handling time h_i. Execution of job i has to start within a pre-specified time window $[r_i, d_i]$. We have a vehicle that can move on the arcs of the graph, at unit speed, and that has to execute all jobs within their respective time windows. We consider three different problems in this class of *vehicle routing problems*. The first problem is to find minimum time cost routes between all pairs of nodes in a network; This is called the all pairs shortest routing(APSR) problem. The second problem is Single vehicle routing problem in which a vehicle services all locations in a network in minimum amount of time. The general problem is $\mathcal{NP}$-complete but $O(n^2)$ time algorithms have been developed when the underlying network is line and there is no time cost in servicing a location, and all the time windows are unbounded at either their lower or upper end. In case, time windows are unbounded at lower end we call dSVRPTW-line problem, and if they are unbounded at upper end we call rSVRPTW-line problem. We show that under the same conditions, we can reduce this problem to the shortest path problem in layered graph and hence obtain a parallel algorithm. We finally consider Traveling repairman problem which is similar to Single vehicle routing problem except we need to minimize the sum of waiting of all locations. The general problem in this case is also $\mathcal{NP}$-complete. But, there is an $O(n^2)$ time algorithm, when the network is line, handling times are zero and there are no time bounds associated with locations. We call this TRP-line problem.

2. Coalescing operations in real-time systems

In real-time systems, an object usually makes scheduling decisions in order to maximize the system performance. There are many ways to make scheduling decisions. *Operation*

Coalescence is one of them [9]. The idea is that some objects may be able to handle several distinct requests at the same time. For example, suppose a stack object is implemented in a system and the public or primitive operations defined for the stack object are: *push*, *pop*, *new* and *top*. We may provide a coalesced operation *double push*, which takes two elements and pushes both of them onto the stack at the same time. Whenever two consecutive *push* operations are found in front of the request queue for the stack object, the object scheduler can invoke the coalesced operation *double-push* instead. We present an $\mathcal{NC}$ algorithm for coalescing operations with precedence constraints in real-time systems when the number of jobs is two.

3. All pairs longest path problem

The all pairs longest path (APLP) problem is to compute the longest path between all pairs of vertices in a directed graph. The all pairs longest distance (APLD) problem is defined similarly, the word "path" is replaced by "distance". We show that the *all pairs shortest path* algorithms proposed by Takaoka [40] can be easily modified to find all pairs longest paths in directed acyclic graphs. The first parallel algorithm solves the APLD problem for a directed acyclic graph with unit edge costs. The second parallel algorithm solves the APLP problem, and consequently the APLD problem, for a directed acyclic graph with non negative costs (real numbers) in $O(\log^2 n)$ time with $o(n^3)$ sub-cubic cost. The results of this problem directly affect the complexities of scheduling interval ordered tasks problem described next.

4. Scheduling interval ordered tasks

The problem of scheduling tasks is to schedule a set of tasks, to a set of processors satisfying the precedence constraint. In this thesis, we consider the problem of scheduling n unit time length tasks on m identical processors or machines for the case, where the precedence constraint is an interval order. The precedence constraint is represented by a partial order. An interval order is a partial order, whose incomparability graph is a chordal graph [32]. (Refer Chapter 2 for definitions of incomparability and chordal graphs). We present an $\mathcal{NC}$ algorithm for this problem. Our algorithm constructs the same schedule as the one produced by the sequential algorithm.

1.2 Model of Computation

To analyze algorithms, we need an abstract model of computation on which all the cost and space requirements of the algorithms of the problem can be expressed and compared against existing ones for answering question of efficiency. In this thesis, we use the Parallel Random Access Machine (PRAM) as the model of computation. PRAM is the parallel analogue of the unit cost sequential random access machine (RAM). In this model there are N processors numbered from 0 to $N-1$. All the processors have access to a memory consisting of cells numbered from 0. Further, the processors are assumed to be aware of their number which is also called their *id*. Depending on the way, simultaneous access to a same memory location by more than one processor is allowed, the PRAM family is classified into 3 classes as described below.

- *Exclusive Read Exclusive Write (EREW):* In this model, at any particular time instant, no more than one processor is allowed to either read from or write into the same memory location. The algorithm designer should write algorithms in such a way that, conflicts never occur.
- *Concurrent Read Exclusive Write (CREW):* In this model, simultaneous read of a memory location by more than one processor is allowed. But, simultaneous write to the same memory location is forbidden.
- *Concurrent Read Concurrent Write (CRCW):* This model allows both simultaneous reading as well as writing a memory location by more than one processor. According to the resolution method of concurrent write, this is further sub-classified into the following models.
  - *COMMON:* All processors writing to a same location should write the same value.
  - *PRIORITY:* The smallest numbered processor succeeds in the write.
  - *ARBITRARY:* One processor is guaranteed to succeed but no commitment is made as to which processor succeeds.
  - *MINIMUM:* The processor having minimum value succeeds in the write.

A model is said to *self-simulating*, if an algorithm which takes $O(t)$ time with p processors, can also be implemented on that model in $O(rt)$ time with $\frac{p}{r}$ processors ($r >$

1) processors. EREW, CREW, common-CRCW models are self-simulating models[26].

A parallel algorithm that runs in time $O(t(n))$ using $p(n)$ processors is said to have a cost of $c(n) = O(t(n)p(n))$. If this cost equals the cost of the best known sequential algorithm, then the parallel algorithm is said to be work optimal [26].

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we review some algorithmic techniques, data structures, basic definitions, notations and complexity classes etc., which are are used in later chapters.

In Chapter 3, we discuss the algorithms for vehicle routing problems. We present serial algorithms for the all pairs shortest routing problem, the TRP-line problem, and parallel algorithms for the all pairs shortest routing problem, the dSVRPTW-line problem, the rSVRPTW-line problem, and the TRP-line problem. In Chapter 4, we present parallel algorithms for coalescing operations with precedence constraints in real-time system.

In Chapter 5, we give parallel algorithms for finding all pairs longest path in directed acyclic graphs. In Chapter 6, we present a parallel algorithm for scheduling interval ordered tasks.

In Chapter 7, we offer some concluding remarks, and state some of the possible extensions to our work.

Chapter 2

Preliminaries

In this chapter, we describe some basic concepts, those we frequently use in subsequent chapters. In section 2.1, we review some basic parallel algorithmic techniques, and in Section 2.2 we give some basic definitions. We explain the shortest path algorithm for Layered directed acyclic graph in Section 2.3, and Fibonacci heap data structure in Section 2.4. We give an introduction to asymptotic notation and some complexity classes of problems in Section 2.5.

2.1 Basic Parallel algorithmic techniques

Prefix Computation. Consider the sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ from a set S, on which a binary *associative* operator '$\otimes$' is defined. The problem of *prefix computation* is defined as finding n "sums" $s_i = x_1 \otimes x_2 \otimes \ldots x_i, 1 \leq i \leq n$.

This problem is solved by organizing the computation in the form of a *balanced binary tree*. Prefix computation problem can be solved in $O(\log n)$ time, with linear work $O(n)$ on a EREW PRAM [28]. Prefix computation is the most frequently used subroutine in parallel algorithms. We use the following variations of prefix computation in the later chapters.

- *Prefix sum:* Considering the binary associative operator to be the usual summation operator $(+)$, we get the prefix sum. The sum of n numbers is given by s_n.
- *Prefix maxima:* Considering the binary associative operator to be the usual maximum operator we get the prefix maxima. The maximum of n numbers is given

by s_n.

Generalized Prefix Computation

Definition 1 [37] *Let $f[m]$ and $y[m]$ be given sequences of elements for integers $1 \leq m \leq n$. Again, let us suppose, there is an arbitrary binary associative operator '$\otimes$' defined on the elements $f[\ ]$, and the elements $y[\ ]$ can be compared by a linear order '$<$'. Then, the problem is to compute the sequence of general prefixes:*
$E[m] = f[j_1] \otimes f[j_2] \otimes \ldots \otimes f[j_k]$ where $j_1 < j_2 < \ldots < j_k$ and $\{j_1, j_2, \ldots, j_k\}$ is set of indices $j < m$ for which $y[j]$ '$<$' $y[m]$, where m is each index from 1 to n.

Generalized prefix computation can be done in $O(\log n)$ time with n processors on a CREW PRAM [37].

Merging. Given two sorted arrays $A[0..n-1]$ and $B[0..n-1]$, the problem is to *merge* them into a single array. If *comparison* is the only allowed operation on the elements, then merging can be done in $O(\log \log n)$ time using $\frac{n}{\log \log n}$ processors on a CREW PRAM [26].

Nearest larger larger problem. Given an integer array $A[1..n]$, the problem is to find the nearest larger $\frac{n}{\log n}$ processors on a CREW PRAM [6].

2.2 Basic Graph Definitions

Definition 2 [32] *The* incomparability graph *of a partial order $P = (V, A)$ is a graph $G = (V, E)$, where $(v, u) \in E$, iff (v, u) and $(u, v) \notin A$. The complements of incomparability graphs have a* transitive orientation, *i.e. an assignment of direction to either of the edges such that resulting directed graph is a partial order.*

Definition 3 [20] *A* chordal graph *is one in which, any circuit $[v_1, \ldots, v_k]$, $k \geq 4$, possesses a chord, i.e. an edge (v_i, v_j) with $j \neq i \pm 1 (\text{mod } k)$.*

Definition 4 [20] *An* interval graph *$G = (V, E)$ is one whose nodes are closed intervals on the real line, and $(v, u) \in E$ iff $v \cap u \neq o$. It is well-known that interval graphs are chordal.*

2.3 Shortest path algorithm for Layered Directed Acyclic Graph

An n Layered directed acyclic graph (LDAG) is a graph with vertices lying on layers; the edges will be from nodes at layer i to $i+1$ only. A simple example of layered graph is grid graph as shown in Figure 1.

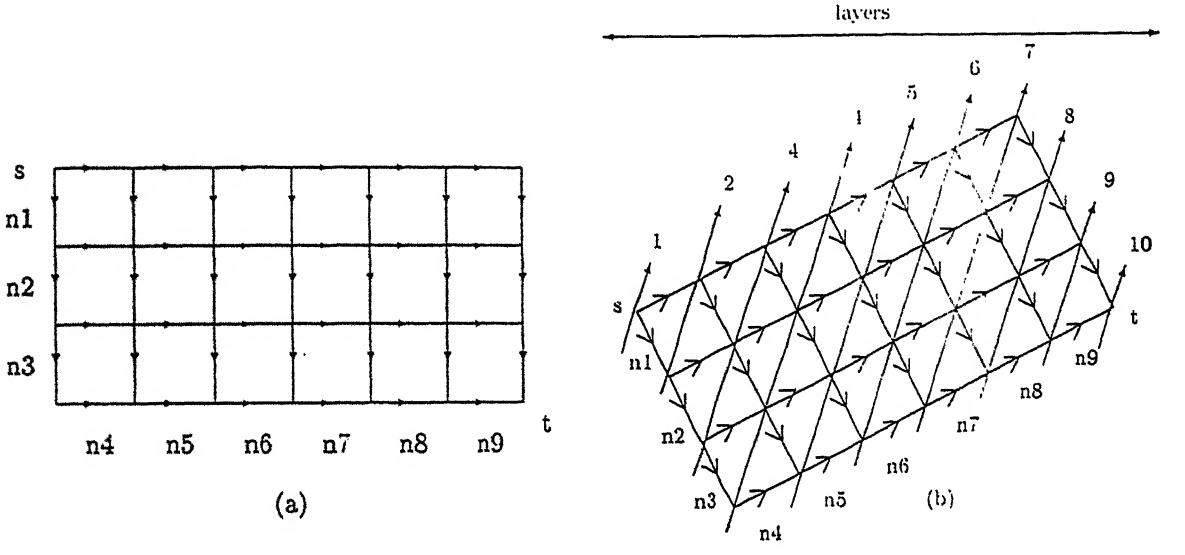


Figure 1: An example of Layered graph

The LDAG shown in Figure 1(a) can be re-drawn as in Figure 1(b). The graph has l rows, $(n-l)$ columns, and $n-1$ layers and maximum number of nodes in any layer is $b = min(l, n-l)$. Edges from nodes at layer i lead to nodes at layer $i+1$ only. If adjacency list is used for storing the edges, the edges to next level can be stored in an array of length $2b$. An element (i,j) will be in layer $(i+j)$, and it is in j^{th} position in array if $1 \leq i+j \leq l$, or in position $l-i$, otherwise.

For finding the shortest path between s and t, the vertices in alternate layers are removed iteratively. The number of layers in each iteration reduces by half. Thus, the algorithm takes $O(\log n)$ iterations. If there is an edge from nodes 'x' at layer $i-1$ to node 'y' at layer i, and if there is another edge from node 'y' to node 'z' at layer $i+1$, then the two edges can be replaced by a single edge from node 'x' to node 'z' with new weight as sum of the weights of original edges. Saxena[35] has shown that, we can find the shortest path in $O(\log n * \log b)$ time, with $O(nb^2/\log b)$ processors on

a CREW PRAM, and in $O(\log n * \log\log b)$ time with $O(nb^2/\log\log b)$ processors on a common-CRCW PRAM.

2.4 Fibonacci heaps

In this section, we describe the Fibonacci heap(F-heap) data structure described by Fredman and Trajan [17].

Definition 5 Heap *is an abstract data structure consisting of set of items, each with a real-valued key.*

It is associated with the following operations operations defined on it.

MakeHeap():	return a new empty heap.
Insert(i, h):	insert a new item i, with a predefined key into heap h.
FindMin(h):	return an item of minimum key in heap h.
DeleteMin(h):	delete, and return an item of minimum key from heap h.
Meld(h_1, h_2):	return the heap formed, by taking the union of item disjoint heaps h_1, h_2. This operation destroys both h_1 and h_2.
DecreaseKey(δ, i, h):	decrease key of item i in heap h, by subtracting the non-negative real number δ. This operation assumes that, the position of i in h is known.
Delete(i, h):	delete arbitrary item i from h. This operation assumes that, the position of i in h is known.

Definition 6 Fibonacci Heap *is a collection of item-disjoint heap ordered trees.* $\mathrm{Rank}(x)$ *denotes the number of children of a node* x. *In the* Data Structure *F-heap, each node has a field to store its rank. Moreover, a node contains a bit, to mark/unmark that node. All nodes are initially unmarked.*

Each node has a pointer to its parent, and a pointer to one of its children. Children of any node are doubly linked in a circular linked list. All roots of trees are also doubly linked in a circular linked list. There is a pointer to the root containing the item of minimum value. F-heap always has to maintain two invariants:

- No two roots can have the same rank
- No non-root can loose two children

To maintain the first invariant we keep an array root[ ] of pointers; root[i] points to root of rank[i], if one exists, otherwise it points to null. Basic heap operations with F-heap are as follows:

MakeHeap(): return a pointer to null.

FindMin(h): return the item in the minimum node.

Meld(h_1, h_2): combine root lists of h_1, h_2 in a single list, and return the root containing the smaller key. Carry out linking step described later.

Insert(i, h): create a new heap containing item i, and replace h by melding h with the new heap. Carry out linking step described later.

Delete(i, h): if item is not the minimum node, then
a) Form a new list of roots by concatenating the list of children(x) with the original list of roots. Make $rank(x)$ equal to zero; effectively, remove all its children form itself. Carry out linking step described later.
b) Call the procedure for cutting the edge, joining x to its parent.
c) Remove x from the list of roots, and destroy node x.

DecreaseKey(δ, i, h): decrease key of item i, if i was the root. check if i is now the minimum node or not. Otherwise, if heap order is violated, call the procedure for cutting the edge, joining i to its parent

Procedure for cutting the edge, joining x to its parent. Insert x in the doubly connected linked list of roots, and check for cascading cuts as follows:

- if $parent(x)$ is unmarked, then mark it and decrease its rank by one.
- if $parent(x)$ is marked, then first decrease its rank by one, and then recursively call the procedure for cutting the edge joining $parent(x)$ to $parent[parent(x)]$.

Thus, if a node looses two of its children through cuts, we cut the edge joining, x to its parent (cascading cut), making x a new node. All these operations can be done in $O(1)$ amortized time[1]. DeleteMin(h) is the only operation. that cannot be done trivially in $O(1)$ time. It takes $O(\log n)$ amortized time[1]. The algorithm for this is as follows:

1. Remove the minimum node say x, from h.

[1]Amortized complexity of an operation is $O(g(n))$ if for a sequence k (sufficiently large, $k \geq n$) operations, the total time required by these operations is $O(kg(n))$ [2].

2. Concatenate the lists of children(x) with the list of roots of h other than x, and repeat the *Linking step*, till it no longer applies.
3. Form a list of remaining roots, in the process finding minimum node.
4. Save the value stored in item x, destroy item x, and return the value.

Linking step. Find any two trees with the same rank and link them. If x is made child of y, then unmark(x).

2.5 Analyzing Algorithms

2.5.1 Order Notation

In this section, we describe various notations we will be using to express time, processor and work complexities of algorithms [14].

1. The notation $f = \Theta(g)$ (read as "f is theta of g") is used, iff there are positive constants C_1, C_2, and a positive integer N such that, $|C_1 * g(n)| \leq |f(n)| \leq |C_2 * g(n)|$, for all $n > N$.
2. The notation $f = O(g)$ (read as "f is oh of g") is used, iff there is a positive constant C and a positive integer N such that, $0 \leq |f(n)| \leq |C * g(n)|$, for all $n > N$.
3. The notation $f = \Omega(g)$ (read as "f is omega of g") is used, iff there is a positive constant C, and a positive integer N such that, $0 \leq |C * g(n)| \leq |f(n)|$, for all $n > N$.
4. The notation $f = o(g)$ (read as "f is little-oh of g") is used, iff there is a positive constant C, and a positive integer N such that, $0 \leq |f(n)| < |C * g(n)|$ for all $n > N$.
5. The notation $f = \omega(g)$ (read as "f is little-omega of g") is used, iff there is positive constant C, and a positive integer N such that, $0 \leq |C * g(n)| < |f(n)|$, for all $n > N$.

2.5.2 Complexity classes

Whenever a new problem is confronted, the first question that arises is "Can it be solved in polynomial time sequentially?". If the answer is yes, then the second question will

be "Can it be solved in poly logarithmic time or constant time in parallel?". To answer these questions, the algorithm designer needs to have some basic understanding about the different classes of problems. In this section, we define some complexity classes of problems.

Definition 7 [4] **Class $\mathcal{NC}$.** *A problem is said to be in class $\mathcal{NC}$, if there exists an algorithm to solve the problem in poly logarithmic time on a PRAM, using polynomial number of processors.*

Definition 8 [4] **Class Random $\mathcal{NC}$.** *A problem is said to be in class random $\mathcal{NC}$ ($\mathcal{RNC}$), if there exists a* random *algorithm to solve the problem in poly logarithmic time on a PRAM, using polynomial number of processors.*

Definition 9 [4] **Class $\mathcal{P}$.** *A problem is said to be in class $\mathcal{P}$, if there exists a sequential algorithm to solve the problem in polynomial time.*

Definition 10 [4] **Class $\mathcal{P}$-complete.** *A problem is said to be $\mathcal{P}$-complete. if it satisfies the following conditions.*

1. *It is in Class $\mathcal{P}$.*
2. *It can be solved in poly logarithmic time on a PRAM, iff all $\mathcal{P}$-complete problems can be solved in poly logarithmic time on a PRAM.*

Definition 11 [24] **Class $\mathcal{NP}$.** *A problem is said to be in class $\mathcal{NP}$, if there exists a* non deterministic *sequential algorithm for solving the problem in polynomial time.*

Definition 12 [24] **Class $\mathcal{NP}$-complete.** *A problem is said to be $\mathcal{NP}$-complete, if it satisfies the following conditions.*

1. *It is in Class $\mathcal{NP}$.*
2. *It is solved in polynomial time, iff all the $\mathcal{NP}$-complete problems are solved in polynomial time sequentially.*

Definition 13 [24] **Class $\mathcal{NP}$-hard.** *A problem is said to be $\mathcal{NP}$-hard. if it is solved in polynomial time then, all $\mathcal{NP}$-complete problems are solved in polynomial time.*

We can easily observe that all $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard but, all $\mathcal{NP}$-hard problems are not $\mathcal{NP}$-complete. Garey and Johnson[18] provide a comprehensive treatment to $\mathcal{NP}$-complete, $\mathcal{NP}$-hard problems.

Chapter 3

Parallel Algorithms For Vehicle Routing Problems

3.1 Introduction

Vehicle routing problems involve navigation of one or more vehicles through a network of locations with each *location* (or *node*) being serviced. Locations are associated with handling times as well as time windows during which they are active. The handling time is the time it takes to service that node. The time window can be specified as a release time - deadline pair where these are the earliest and latest times that the node can be serviced. The arcs connecting locations have time costs associated with them. A vehicle navigating through a network travels through a sequence of locations; such a sequence is called route. The length of route is the sum of handling times of each node on the route and the travel times along each edge on the route. However, if the vehicle arrives at a node in the route before the release time of that node, it must wait until the release time has elapsed before proceeding through the node; these wait times are added to routes length. If the vehicle arrives at a node past its deadline time the route is not feasible.

In this chapter, we develop parallel algorithms for *all pairs shortest routing*(APSR), *Single vehicle routing problem*(SVRP) and *Traveling repairman problem*(TRP). In the first problem, the *all pairs shortest routing problem*, we compute the shortest route between all pairs of locations. Our algorithm runs in time $O(\log^3 n)$ time on a CREW PRAM using $O\left(\frac{n^4}{\log^2 n}\right)$ processors. The best known previous algorithm given by Gupta

et al. [21] runs in $O(\log^3 n)$ time using $O(n^4)$ processors on a CREW PRAM.

A network is said to be a line, if all locations in the network lie on a line. In the second problem, the *single vehicle routing problem* we compute the shortest route involving all locations starting from a particular locations called *Depot*. Here the network is line, and all the locations will have either release time or deadline time. Our algorithm for these problems take $O(\log^3 n)$ time using $O\left(\frac{n^4}{\log^2 n}\right)$ processors on a CREW PRAM. The best known previous algorithms given by Gupta *et al.* run in $O(\log^3 n)$ time with $O(n^8)$ processors on a CREW PRAM. In Gupta *et al.* [21] they have wrongly stated that their algorithms for the APSR problem and the SVRP problems run in $O(\log^2 n)$ time[2].

The third problem, the *traveling repairman problem* is similar to the second one but, here we try to find a route involving all locations and the sum of waiting times at all locations is to be minimum. In this problem, the network is a line and locations don't have any time windows associated with them. We show that this problem can be reduced to shortest path algorithm in layered graph and give an $O(\log^2 n)$ time parallel algorithm with $O\left(\frac{n^3}{\log n}\right)$ processors on a CREW PRAM.

These problems arise in the service sector in applications such as garbage collection and postal delivery, in the commercial sector in the transportation of goods through road and rail, and in the industrial sector in material handling systems in manufacturing. In addition, these problems arise in experimental applications such as automatic vehicle routing and robot arm movement[21].

The problem of navigating a vehicle in a city arises in a variety of guises, each street has a time cost for travel on the street, each intersection has an expected delay and some intersections or streets might be open only at certain times, vehicle routing problem requires a vehicle to travel as quickly as possible from one location to another respecting these timing constrains. Alternatively, the vehicle may be required to service a number of locations in the city in the minimum time, again respecting the timing constraints [21].

The Vehicle routing problem (VRP) involves the design of a set of minimum cost routes, originating and terminating at central depot, for a fleet of vehicles which services a set of customers with known demands [36]. Each customer is serviced exactly once

[2]They have stated that composition of two tables or finding minimum of two tables requires $O(1)$ time. But, these operations require $O(\log n)$ time.

and furthermore, all the customers must be assigned to vehicles such that the vehicle capacities are not exceeded. Bodin *et al.* [8] provide a comprehensive survey of the VRP and its variations which also describes the many practical occurrences of these problems.

In VRP with time windows (VRPTW), the above issues have to be dealt with under the added complexity of time windows. Time windows specify the deadlines and earliest service times of customer. The service of customer can begin within the time window defined; service involving pickup and/or delivery of goods. These time windows can be soft or hard. In case of hard windows the service has to start within the time window. However, in case of soft windows the service time can violate time window with some penalty. Solomon *et al.* [36] provide a survey of time constrained routing and scheduling problems.

	Zero processing times	General processing times
No release times or deadlines	Trivial	Trivial
Release times only	$O(n^2)$[33]	$\mathcal{NP}$-complete[41]
Deadlines only	$O(n^2)$[41]	?
General time windows	Strongly $\mathcal{NP}$-complete[41]	Strongly $\mathcal{NP}$-complete[19]

Table 1: The complexity of special cases of SVRPTW-line(n is the number of jobs)

SVRP is a case of VRP when then there is only one vehicle and it is $\mathcal{NP}$-complete even if the inter point distance metric is restricted to Euclidean [31]. Introducing time constraints on the problem (SVRPTW) can only make it harder [34]. There are some polynomial time algorithms when we restrict underlying network to a straight line and the vehicle is incapacitated. Psaraftis *et al.*[33] study this variant where nodes have zero handling times and all time windows have only release time constraints. Tsitsiklis [41] studies a similar problem except the time window for each node has only a deadline constraint. We call these the SVRPTW-line problem with release (rSVRPTW) and the SVRPTW-line problem with deadline (dSVRPTW) respectively. In both cases, the authors provide an $O(n^2)$ time algorithm for their respective problems based on a dynamic-programming formulation. Tsitsiklis in addition shows that the SVRPTW-line

problem with general time windows but zero handling times is $\mathcal{NP}$-hard. He also shows that SVRPTW-line with a release time constraint with non-zero handling time is $\mathcal{NP}$-hard. Karuno *et al.* [27], consider the case in which the underlying network is a tree, time windows have only release time constraints and each node is allowed a non-zero handling time. They show that the problem is $\mathcal{NP}$-hard and provide an approximation algorithm with a performance ratio of two.

	Zero processing times	General processing times
No release times or deadlines	$O(n^2)$[1]	?
Release times only	?	strongly $\mathcal{NP}$-complete[29]
Deadlines only	$\mathcal{NP}$-complete[1]	$\mathcal{NP}$-complete[41]
	pseudo polynomial	$\mathcal{NP}$-complete[41]
General time windows	Strongly $\mathcal{NP}$-complete[41]	Strongly $\mathcal{NP}$-complete[19]

Table 2: The complexity of special cases of TRPTW-line(n is the number of jobs)

Traveling repairman problem(TRP) is a variation of the well known Traveling salesman problem, in which instead of minimizing the total completion time for salesman tour, one tries to minimize the sum of the waiting times of locations or customers [1]. TRP captures the waiting costs of a service system from the customers point of view and it can be used to model numerous types of service systems. While the general problem is $\mathcal{NP}$-complete [1], some progress has been made when the network is a straight line. Afrati *et al.* [1] have given a $O(n^2)$ time algorithm when the handling time of location is zero and there are no time bounds associated with locations[41]. We call this as TRP-line problem.

In Section 3.2, we give definitions and formal description of problems considered. Section 3.3 provides a description and analysis of the parallel algorithm for APSR problem. In Section 3.4, we describe algorithms for rSVRPTW-line and dSVRPTW-line problems. In Section 3.5, we give an algorithm for TRP-line problem.

3.2 Preliminaries

The underlying network can be represented as a graph $G(V, E)$, where the set of nodes V are locations, and the set of edges E are links between locations. Each edge (u, v) has weight $t(u, v)$ associated with, the time required for the vehicle to traverses this edge called *travel time*. Each node v will have *handling time* $h(v)$ that denotes the time required for vehicle to service the node and a time window associated with it. The vehicle can pass through the node on the way to some other node on the route or actually handle the node during its time window. The time window can be specified as *release time* $r(v)$, and *deadline time* $d(v)$ pair, where these are the earliest and latest times that the node can be serviced. Further, we assume $0 \leq r(v) \leq d(v) < \infty$ for all nodes, v and that all weights are positive, and integral. The closed interval $[r(v), d(v)]$ denote the *time window* of v.

A network G is called *line*, if its nodes can be ordered as $v_1, v_2, \ldots, v_n$ such that there is an edge from v_i to v_{i+1}, for $1 \leq i < n$, and there are no other edges. The travel time definition is extended to be the length of shortest path from u to v. A route is a set of nodes serviced in the order specified in the sequence. A route is *feasible*, if its cost is defined; otherwise it is *infeasible*. The cost of route at a particular starting time is defined as, if the vehicle can traverse through the nodes (in the route) with every node being serviced before its deadline time. The cost of route can be formally defined as follows.

Definition 14 [21] *A* route *is a sequence of nodes* $v_0, v_1, \ldots, v_s$ *such that if* $i \neq j$ *then* $v_i \neq v_j$. *The* cost *of the route* $C_T(v_0, v_1, \ldots, v_s)$, *is a function of the starting time* T, *and is defined inductively:*

1. If $T > d(v_0)$, then $C_T(v_0)$ is undefined;
2. If $T < r(v_0)$ then $C_T(v_0) = r(v_0) + h(v_0)$;
3. If $r(v_0) \leq T \leq d(v_0)$, then $C_T(v_0) = T + h(v_0)$;
4. If $C_T(v_0, v_1, \ldots, v_i)$, is undefined then $C_T(v_0, v_1, \ldots, v_{i+1})$, is undefined;
5. If $C_T(v_0, v_1, \ldots, v_i) + t(v_i, v_{i+1}) > d(v_{i+1})$, then $C_T(v_0, v_1, \ldots, v_{i+1})$ is undefined;
6. If $C_T(v_0, v_1, \ldots, v_i) + t(v_i, v_{i+1}) < r(v_{i+1})$, then $C_T(v_0, v_1, \ldots, v_{i+1}) = r(v_{i+1}) + h(v_{i+1})$;
7. If $r(v_{i+1}) \leq C_T(v_0, v_1, \ldots, v_i) + t(v_i, v_{i+1}) \leq d(v_{i+1})$, then

$C_T(v_0, v_1, \ldots, v_{i+1}) = C_T(v_0, v_1, \ldots, v_i) + t(v_i, v_{i+1}) + h(v_{i-1})$.

The cost of route is *optimal*, if there is no route with smaller cost from v_0 to v_n. It is *covering*, if every node of G appears in the route. The problems considered in this chapter are defined as follows.

Definition 15 *The* all pairs shortest routing problem*(APSR) determines, for each pair of nodes u and v in G, the optimal route between u and v, for each possible starting time of the vehicle, starting from node u, and servicing all nodes in the route within their respective time windows.*

Definition 16 *The* single vehicle routing problem with time window constraints*(SVRPTW) on G finds the optimal covering route of G that starts at depot($S(G)$) at time 0, and ends at $T(G)$, where $T(G)$ can be any node.*

Lemma 1 [21] *SVRP is $\mathcal{NP}$-hard even for $h(v), r(v) = 0$, and $d(v) = \infty$, for every $v \in V^*(G)$.*

The proof of this lemma follows by a reduction from TSP. In this chapter, we only consider SVRPTW problem for which $h(v) = 0$ with underlying network as line. Further, we discuss only SVRPTW-line problems having only release times (i.e. $r(v) = 0\ \forall v$), and SVRPTW-line problems having deadlines (i.e. $h(v) = \infty\ \forall v$) These are called rSVRPTW-line, and dSVRPTW-line respectively.

Definition 17 *The* traveling repairman problem*(TRP) on G finds the covering route of G that starts at depot($S(G)$) at time 0, and ends at $T(G)$ having minimum sum of waiting times of nodes The waiting time of node is the difference between release time, and the actual time at which vehicle services the node.*

In this chapter, we discuss TRP-line problem for which all nodes in the network have zero handling time, and there is no time windows associated with them.

3.3 All pairs shortest routing problem

The *handling time* term can be eliminated by simply adding $h(v)$ of node v to the travel time on each edge out of v and assume that the networks do not have handling times [21]. In Section 3.3.1, we describe a sequential algorithm for the problem of finding a shortest route from a node S to a node T using Fibonacci heap data structure given by Fredman and Trajan [17]. This in-fact finds the shortest route from node S to all

other nodes in G. This is called S-T routing problem [21]. The algorithm described has $O(m + n \log n)$ time complexity, where m is the number of edges and n is the number of nodes in the network. This is an improvement over the $O(n^2)$ time algorithm given by Gupta *et al.* [21]. In Section 3.3.2, we give an parallel algorithm for APSR problem.

3.3.1 Sequential algorithm for the APSR problem

Let $G(V, E)$ be a network with time windows specified at each node $u \in V$ and travel times specified on edge $(u, v) \in E$. Let n be number of nodes and m be the number of edges. Gupta *et al.* have described an $O(n^2)$ algorithm for this problem. Our algorithm proceeds in the manner analogous to that of Fibonacci heap(F-heap) implementation of Dijkstra's algorithm.

1. Un-label all the vertices $v \in G$.
2. Cost function T is defined as follows
   The cost function for S is $T(S) = 0$, and for all other vertices u is $T(u) = \infty$, $prev(u) = S$
3. for all neighbors v of S, we update T as follows
   $T(v) = \max\{r(v), T(S) + t(S, v)\}$
   if $T(S) + t(S, v) > d(v)$ then $T(v) = \infty$
4. while($\exists v \in V$ which is un-labeled)
   (a) Select an un-labeled vertex v, having minimum $T(v)$, Mark it.
   (b) For each edge (v, w), $T(w) = \min\{T(w), \max\{r(w), T(v) + T(v, w)\}\}$. If $T(v) + T(v, w) > d(w)$ then $T(w) = \infty$.

Lemma 2 *Sequential algorithm for S-T routing problem takes $O(m + n \log n)$ time.*

Proof: If all un-labeled vertices are kept in a F-heap then Step 4(a) requires a delete minimum operation, and Step 4(b) require a decrease key operation; in that case we also make $prev(w) = v$. Moreover, for every edge (u, v), Step 4(b) is performed exactly once. Thus Step 4(b) will be performed exactly m times. Moreover, Step 4(a) will be performed exactly once for each vertex. Thus, we have n minimum deletion operations and and m decrease key operations on a F-heap data structure. A delete minimum operation takes $O(\log n)$ amortized time and a decrease key operation requires $O(1)$ amortized time [17]. Thus the total time taken by our algorithm is $O(m + n \log n)$. ∎

Theorem 1 *Sequential algorithm for APSR problem takes $O(nm + n^2 \log n)$ time.*

Proof: The proof of the theorem follows from the fact that, APSR problem can be solved by performing S-T routing problem once for each vertex v in G i.e. n times. ∎

3.3.2 Parallel Algorithm for the APSR problem

For the parallel case, we must keep track of significantly more information at each step. Suppose we are trying to compute an optimal S-T route for a pair of nodes S and T of G. Let $S, v_1, \ldots, v_k, T$ be such an optimal S-T route. Then, by starting at $v_{\frac{k}{2}}$ and S simultaneously, we can look for optimal paths from S to $v_{\frac{k}{2}}$ starting at time 0 and from $v_{\frac{k}{2}}$ to T starting at time $C(S, v_1, \ldots, v_{\frac{k}{2}})$. We can now recursively solve S-$v_{\frac{k}{2}}$ and the $v_{\frac{k}{2}}$-T routing problems.

There are two reasons why we cannot directly apply this technique. First the routes S-$v_{\frac{k}{2}}$ and $v_{\frac{k}{2}}$-T cannot be computed independently - because of time-windows. we need $C(S, v_1, \ldots, v_{\frac{k}{2}})$ to compute the optimal $v_{\frac{k}{2}}$-T route. Second, we do not know which node of G is $v_{\frac{k}{2}}$. Trying all possibilities may result in an algorithm requiring $O(n^{\log n})$ processors. We compute the optimal route as a function of time to address the first issue, and use a variant of pointer doubling to handle the second [21].

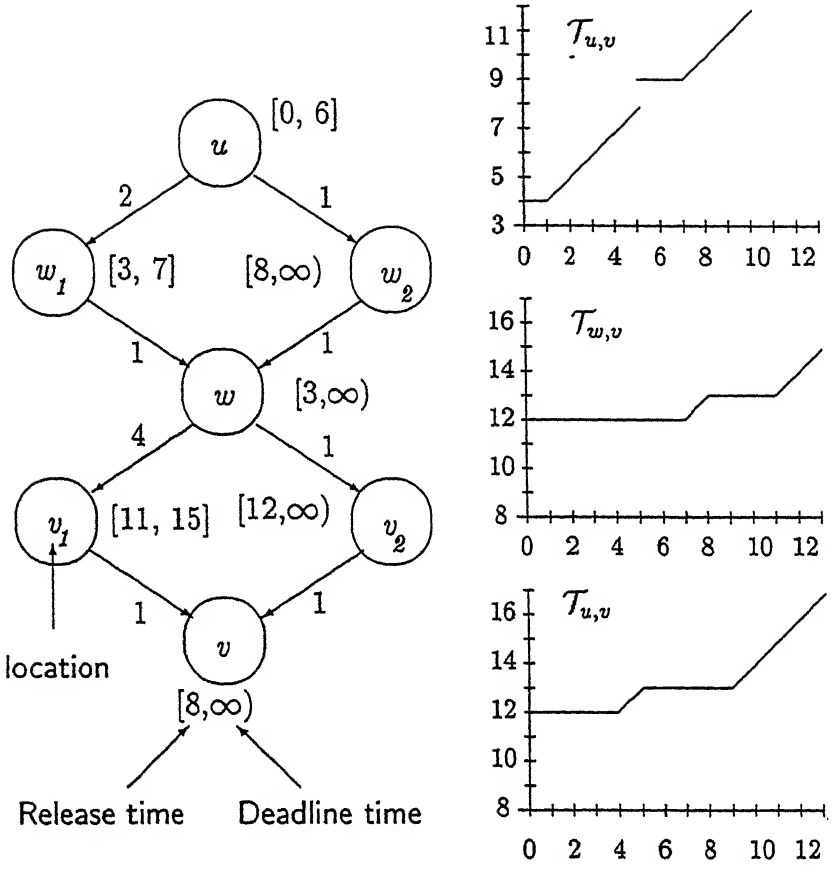
3.3.2.1 Cost Vectors

A function $\mathcal{T}_{u,v}$ is associated with every pair $(u, v) \in E(G)$; $\mathcal{T}_{u,v}(t)$ denotes the cost of an optimal u-v route starting at time t.

For the S-T routing problem, the optimal cost of $v_{\frac{k}{2}}$-T route is computed as a function of $\mathcal{T}_{S,v_{\frac{k}{2}}}$ and in parallel the cost of the optimal S-$v_{\frac{k}{2}}$ route is also computed. By evaluating the function at the actual cost. we have the cost of optimal S-T route.

Definition 18 [21] *For every $(u, v) \in E(G)$ and $t \geq 0$, let $\mathcal{P}_{u,v}(t)$ denote an optimal cost route from u to v when the vehicle starts from u at time t. The cost vector of the optimal route from u to v is a function $\mathcal{T}_{u,v} : N \rightarrow N$ such that $\mathcal{T}_{u,v}(t)$ is the cost of $\mathcal{P}_{u,v}(t)$.*

The cost vector is monotonic in t since the vehicle must wait at nodes whose time windows are not yet open. Figure 2 shows an example of cost function.



$\mathcal{T}_{u,w}$

interval	*cost*	*route*
[0,1]	4	u, w_1, w
[2,5]	$\tau + 3$	u, w_1, w
[6,7]	9	u, w_2, w
[8,∞)	$\tau + 2$	u, w_2, w

$\mathcal{T}_{w,v}$

interval	*cost*	*route*
[0,7]	12	w, v_1, v
[8,8]	$\tau + 5$	w, v_1, v
[9,11]	13	w, v_2, v
[12,∞)	$\tau + 2$	w, v_2, v

$\mathcal{T}_{u,v}$

interval	*cost*	*route*
[0,4]	12	u, w_1, w, v_1, v
[5,5]	$\tau + 7$	u, w_1, w, v_2, v
[6,9]	13	u, w_2, w, v_2, v
[10,∞)	$\tau + 4$	u, w_2, w, v_2, v

Figure 2: An example Graph G with time windows and travel times. Three tables specify the cost vector as well as optimal routes for each row

Let $\mathcal{P}$ be an optimal route in G during all times in a time interval $\mathcal{I} = [a, b]$. Let $W > 0$ be the sum of the waiting times, when a vehicle follows $\mathcal{P}$ starting from u at time a. If $b > a + W$, $\mathcal{P}$ makes a transition at time $a + W$: before this time, a vehicle following $\mathcal{P}$ is forced to wait at one or more nodes but after this time there are no waits. $\mathcal{P}$ is called *transition route* and $a + W$ its *transition time*. In the transition route $\mathcal{P}$, there is at least one one node w such that if the vehicle starts at any time $t < a + W$, it must wait at w and it is called *bottle-neck* node of $\mathcal{P}$ and $bn(\mathcal{P})$ denotes the set of

such nodes [21].

Lemma 3 [21] *Suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ are two transition routes from u to v that are optimal during time intervals $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. If $\mathcal{I}_1$ occurs strictly before $\mathcal{I}_2$ then* $bn(\mathcal{P}_1) \cap bn(\mathcal{P}_2) = \emptyset$

The cost vector $\mathcal{T}_{u,v}$ can be represented by a table satisfying the following properties [21]:

1. each row r of the table consists of $interval(r)$ and $cost(r)$
2. $interval(r)$ is of the form $[a, b]$ or $[a, \infty)$ where $0 \leq a \leq b < \infty$. a and b are lower and upper bound of interval respectively
3. for rows r_1 and r_2, $r_1 < r_2$, all elements of $interval(r_1)$ precede all elements of $interval(r_2)$
4. the union $interval(r)$ over all rows r is the interval$[0, \infty)$
5. $cost(r)$ is a function of the form α or $\tau + \alpha$ where α is a constant τ is a variable
6. if $cost(r) = \alpha$ (respectively $\tau + \alpha$) then for all $t \in interval(r)$. $\mathcal{P}_{u,v}(t)$ has cost α (respectively $t + \alpha$)
7. there is one route $\mathcal{P}_r$ associated with each r such that $\mathcal{P}_r$ is an optimal u-v route when the vehicle starts from u at any time $t \in interval(r)$

and there is no smaller table satisfying the above conditions.

The *complexity* of a cost vector $\mathcal{T}_{u,v}$, $complex(\mathcal{T}_{u,v})$, is the minimum number of rows required to represent it as a table $complex(\mathcal{T}_{u,v}) \in O(n)$ [21].

Lemma 4 [21] *Given two nodes u and v,* $\text{complex}(\mathcal{T}_{u,v}) \leq 4n$

3.3.2.2 Pointer Doubling algorithm

In this section, we describe our parallel algorithm for APSR problem. This algorithm is based on the composition and minimization of cost vectors to compute new cost vectors and it uses the pointer doubling technique, introduced by Fortune and Wyllie [15]; Pointer doubling technique is inherent in such algorithms as computing the transitive closure of an adjacency matrix [12] and algorithms for tree contraction [12]

High level description of algorithm is [21]:

1. For every $(u, v) \in E(G)$ in parallel compute $\mathcal{T}_{u,v}$
2. For every non-edge (u, v), let $\mathcal{T}_{u,v}$ be one row table with interval $[0, \infty)$ and cost

3.4 Parallel algorithms for dSVRPTW-line and rSVRPTW-line Problems

Tsitsiklis [41] and Psaraftis *et al.* [33] have used dynamic programming technique to obtain quadratic time algorithms for dSVRPTW-line, rSVRPTW-line problems respectively. Our algorithms for these problems make use of the parallel algorithm for the shortest path problem for LDAG described in Section 2.3 combined with the dynamic programming formulation of Tsitsiklis and Psaraftis *et al.* We describe the sequential algorithm given by Tsitsiklis in Section 3.4.1 and then in Section 3.4.2, we give our parallel algorithms for dSVRPTW-line problem. Parallel algorithms for rSVRPTW-line problem is similar to dSVRPTW-line. So, we do not discuss it further.

Let G be an instance of dSVRPTW-line. We can fix G to be the network ordered in the form $a_m, a_{m-1}, \ldots, a_1, D, b_1, \ldots, b_p$; where D is the central depot from where the vehicle originates and terminates. Since all the locations are on a line there are edges between two consecutive locations only. Every edge will have cost associated with it; the cost denotes the time taken for the vehicle to travel between the two locations. The optimal feasible routes for G can be assumed to be in a certain canonical form. For ease of exposition, we can assume that the nodes a_0 and b_0, both refer to same node, the depot D [21].

Definition 19 *Let $S = v_1, \ldots, v_{m+p+1}$ be an optimal feasible route that covers G. Then S is* uniform *if for any $k \leq m+p+1, v_1, \ldots, v_k$ is an optimal feasible route that covers the subgraph of G induced by $a_i, a_{i-1}, \ldots, a_1, D, b_1, \ldots, b_j$ for some i and j such that $k = i+j+1$.*

Lemma 9 [21] *If there is a feasible route then there is an optimal feasible route*

Tsitsiklis [41] combined the notion of uniform routes with dynamic programming to obtain a quadratic algorithm for dSVRPTW-line problem.

3.4.1 Sequential algorithm for dSVRPTW-line

Let $v_1, \ldots, v_{m+p+1}$ be a uniform route in G. We can represent the subsequence $v_1, \ldots, v_k$ by the ordered pair (i, j) where a_i and b_j both appear in this subsequence and $i+j+1 = k$. Then, the subsequence $v_1, \ldots, v_k, v_{k+1}$ is represented by either $(i+1, j)$ or $(i, j+1)$ depending on whether v_{k+1} is a_{i+1} or b_{j+1}. More generally, for $0 \leq i \leq m$ and $0 \leq j \leq p$,

∞.

3. Loop $\log n$ rounds

   In parallel for every pair $(u, v) \in E(G)$

   (a) $A = \{w \mid w \in V(G); (u, w), (w, v) \in E(G)\}$

   (b) $\forall w \in A$, compose $\mathcal{T}_{u,w}$, $\mathcal{T}_{w,v}$ to form $\mathcal{T}^w_{u,v}$

   (c) Let $\mathcal{T}_{u,v} = \min\{\mathcal{T}_{u,v}, \min\{\mathcal{T}^w_{u,v} \mid w \in A\}\}$

   (d) If (u, v) is not in $E(G)$ then add edge (u, v) to $E(G)$

Operator of composing $\mathcal{T}_1$ and $\mathcal{T}_2$ can be viewed graphically as follows: For each row r in $\mathcal{T}_1$, calculate the corresponding cost at both ends of the time interval for that row. This cost interval computed becomes the time interval of $\mathcal{T}_2$. Then copy the graph for $\mathcal{T}_2$ to get the graph for $\mathcal{T}_{12}$ for that interval. Putting it more formally, we have the following algorithm.

Algorithm for composing $\mathcal{T}_1$ and $\mathcal{T}_2$.

1. for each row r in $\mathcal{T}_1$, find rows s in $\mathcal{T}_2$ such that cost interval of r overlaps with time interval of s;
2. for each pair r in $\mathcal{T}_1$, and s in $\mathcal{T}_2$ (cost(r) overlaps with interval(s)):

   (a) compute the cost function by substituting the cost(r) in cost(s);

   (b) create a row in $\mathcal{T}_{12}$ with newly computed cost function.

3. merge the rows in $\mathcal{T}_{12}$ which have identical cost function and route:

Algorithm for finding the overlapped intervals.

1. For each row $r \in \mathcal{T}_1$ calculate the corresponding cost at both ends of the time interval. Now we can see that table is of four columns with first two columns representing the time interval (say $\mathcal{T}_1(1), \mathcal{T}_1(2)$) and last two representing the cost interval (say $\mathcal{T}_1(3), \mathcal{T}_1(4)$).

   Let $\mathcal{T}_2(1), \mathcal{T}_2(2)$ are the columns representing lower and upper end of time intervals of $\mathcal{T}_2$.
2. Merge arrays $\mathcal{T}_2(1)$ and $\mathcal{T}_1(3)$ and call it $\mathcal{T}_l$.
3. Form an array I_l where, if $\mathcal{T}_l(i) \in \mathcal{T}_2$ then $I_l(i) =$ rank of $\mathcal{T}_l(i)$ in $\mathcal{T}_2$ else $I_l(i) = 0$.
4. For every $I_l(i)$ such that $\mathcal{T}(i) \in \mathcal{T}_1$ ($I_l(i) = 0$), find the the leftmost time interval of $\mathcal{T}_2$ in which the cost interval of $\mathcal{T}_1$ falls. (Find the nearest larger left hand side

in $\mathcal{T}_l$). Say it r_l. In other words, r_l gives the index of the first row in $\mathcal{T}_2$ which overlaps with cost interval of $\mathcal{T}_1$.

5. merge arrays $\mathcal{T}_1(4)$ and $\mathcal{T}_2(2)$ and call it $\mathcal{T}_r$. Form an array I_r where. if $\mathcal{T}_r(i) \in \mathcal{T}_2$ then $I_r(i)$ = rank of $\mathcal{T}_r(i)$ in $\mathcal{T}_2$ else $I_r(i) = 0$.
6. For every $I_r(i)$ where $\mathcal{T}(i) \in \mathcal{T}_1$ ($I_r(i) = 0$) find the rightmost time interval of $\mathcal{T}_2$ in which the the cost interval of $\mathcal{T}_1$ falls. (Find the nearest larger right hand side in $\mathcal{T}_l$). Say it r_r. In other words, r_r gives the index of the last row in $\mathcal{T}_2$ which overlaps with cost interval of $\mathcal{T}_1$.
7. For every r in $\mathcal{T}_1$, find the value $r_o = r_r - r_l + 1$.
   It represents the number of rows in $\mathcal{T}_2$, whose time interval overlap with cost interval of r.

Since the cost functions are monotonic and time intervals in tables do not overlap, the total size of the composed table is sum of tables that are being composed. If we do a prefix sum of r_o for every row (excluding its own r_o from prefix sum), to get the indices of the each row in the final table constructed.

Lemma 5 [21] *$\mathcal{T}^w_{u,v}$ can be described by a table using at most $complex(\mathcal{T}_{u,v})+complex(\mathcal{T}_{w,v})$ rows.*

$\mathcal{T}^w_{u,v}$ is a cost vector in a sub graph of G. hence by Lemma 4 $complexity$ of $\mathcal{T}$ is at most $4n$; thus some rows of $\mathcal{T}^w_{u,v}$ that can be merged.

An example describing algorithm composition operator

Let us take the tables $\mathcal{T}_{uv}$ and $\mathcal{T}_{vw}$ shown in Figure 2.

1. computation of cost function at both ends of time interval of $\mathcal{T}_{uw}$.

$\mathcal{T}_{vw}$

interval	cost
$[0,1]$	$[4,4]$
$[2,5]$	$[5,8]$
$[6,7]$	$[9,9]$
$[8,\infty)$	$[10,\infty)$

$\mathcal{T}_{wv}$

interval	cost
$[0.7]$	12
$[8,8]$	$\tau + 5$
$[9,11]$	13
$[12,\infty)$	$\tau + 2$

2. $\mathcal{T}_l = \text{merge}(\mathcal{T}_2(1), \mathcal{T}_1(3)) = [0.4, 5, 8, 9.9, 10, 12]$
   $\mathcal{T}_l = [1, 0, 0.2, 3, 0, 0, 4]$
3. $r_l = [1, 1, 3, 3]$ (nearest larger on left hand side for zeros in $\mathcal{T}_l$)
4. $\mathcal{T}_r = \text{merge}(\mathcal{T}_1(4), \mathcal{T}_2(2)) = [4, 7, 8, 8, 9.11, \infty, \infty]$

$T_r = [0, 1, 0, 2, 0, 3, 0, 4]$

5. $r_r = [1, 2, 3, 4]$ (nearest larger on right hand side for zeros in T_r)
6. $r_r - r_l + 1 = [1, 2, 1, 1]$
7. this means first row of $\mathcal{T}_{uw}$ overlaps with one (first) row of $\mathcal{T}_{wv}$,
   second row of $\mathcal{T}_{uw}$ overlaps with two (first, second) rows of $\mathcal{T}_{wv}$,
   third row of $\mathcal{T}_{uw}$ overlaps with one (third) row of $\mathcal{T}_{wv}$,
   fourth row of $\mathcal{T}_{uw}$ overlaps with one (fourth) row of $\mathcal{T}_{wv}$,
8. table $\mathcal{T}_{uv}$ after calculating the cost function, before merging rows.

interval	cost	route
$[0,1]$	12	u, w_1, w, v_1, v
$[2,4]$	12	u, w_1, w, v_1, v
$[5,5]$	$\tau + 8$	u, w_1, w, v_2, v
$[6,7]$	13	u, w_2, w, v_2, v
$[8,9]$	13	u, w_2, w, v_2, v
$[10,\infty)$	$\tau + 4$	u, w_2, w, v_2, v

9. table $\mathcal{T}_{uv}$ after merging.

interval	cost	route
$[0,4]$	12	u, w_1, w, v_1, v
$[5,5]$	$\tau + 8$	u, w_1, w, v_2, v
$[6,9]$	13	u, w_2, w, v_2, v
$[10,\infty)$	$\tau + 4$	u, w_2, w, v_2, v

Lemma 6 *The composition operation* $\{\forall w \in A \mid \mathcal{T}^w_{u,v} = \mathcal{T}_{u,w} + \mathcal{T}_{w,v}$ *can be performed in* $O(\log n)$ *time using* $\frac{n^2}{\log n}$ *processors on a CREW PRAM.*

Proof: Our algorithm for composition of two tables is based on finding prefix sum on inputs of size $O(n)$, merging two arrays of sizes $O(n)$ and finding nearest larger in an array of length $O(n)$. These operations take $O(\log n)$ time using $\frac{n}{\log n}$ processors on a CREW PRAM [28, 26, 6]. So, $O(\log n)$ time with $\frac{n}{\log n}$ processors is sufficient to compose two tables. The composition operation described above needs composition of n pairs of tables. So, it takes $O(\log n)$ time with $\frac{n^2}{\log n}$ processors. ∎

One way of looking at problem of finding minimum of $\mathcal{T}_1$ and $\mathcal{T}_2$ is to find the time intervals of $\mathcal{T}_1$ which overlap with a time interval of $\mathcal{T}_2$ for every row. Then find the minimum cost of the intervals overlapped. Putting it more formally we have the following

algorithm.

Algorithm for finding minimum of two tables $\mathcal{T}_1$ and $\mathcal{T}_2$.

1. for each row r in $\mathcal{T}_1$, find rows s in $\mathcal{T}_2$ such that time interval of r overlaps with time interval of s;
2. for each pair r in $\mathcal{T}_1$, and s in $\mathcal{T}_2$ (such that interval(r) overlaps with interval(s));
   (a) compute the cost function by taking minimum of cost(r), cost(s);
   (b) create a row in $\mathcal{T}_{12}$ with newly computed cost function;
3. merge the rows in $\mathcal{T}_{12}$ which have identical cost function and route.

For finding the overlapped intervals we can follow a similar procedure described for composing two tables algorithm.

An example describing algorithm minimization operator

Let us take the cost vectors $\mathcal{T}_{uw_1w}$ and $\mathcal{T}_{uw_2w}$ of graph in Figure 2.

1. $\mathcal{T}_{uw_1w}$

interval	cost
$[0,1]$	4
$[2,5]$	$\tau+3$
$[6,\infty)$	∞

$\mathcal{T}_{uw_2w}$

interval	cost
$[0,7]$	9
$[8,\infty)$	$\tau+2$
-	-

2. $\mathcal{T}_l = \text{merge}(\mathcal{T}_2(1), \mathcal{T}_1(1)) = [0, 0, 2, 6.8]$
   $T_l = [1, 0, 0, 0, 2]$
3. $r_l = [1, 1. 1]$ (nearest larger on left hand side for zeros in T_l)
4. $\mathcal{T}_r = \text{merge}(\mathcal{T}_1(1), \mathcal{T}_2(1)) = [1, 5, 7, \infty. \infty]$
   $T_r = [0, 0, 1, 0, 2]$
5. $r_r = [1, 1, 2]$ (nearest larger on right hand side for zeros in T_r)
6. $r_r - r_l + 1 = [1, 1, 2]$
7. this means first row of $\mathcal{T}_{uw_1w}$ overlaps with one (first) row of $\mathcal{T}_{uw_2w}$,
   second row of $\mathcal{T}_{uw_1w}$ overlaps with one (first) row of $\mathcal{T}_{uw_2w}$,
   third row of $\mathcal{T}_{uw_1w}$ overlaps with two (first,second) rows of $\mathcal{T}_{uw_2w}$.
8. table $\mathcal{T}_{uw}$ after calculating the cost function (There are no rows to be merged).

interval	cost	route
$[0,1]$	4	u, w_1, w
$[2,5]$	$\tau+3$	u, w_1, w
$[6,7]$	9	u, w_2, w
$[8,\infty)$	$\tau+2$	u, w_2, w

Lemma 7 *The minimization operation* $\mathcal{T}_{u,v} = \min\{\mathcal{T}_{u,v}, \min\{\mathcal{T}^w_{u,v} \mid w \in A\}\}$ *can be performed in* $O(\log^2 n)$ *time using* $\frac{n^2}{\log^2 n}$ *processors on a CREW PRAM.*

Proof: Finding the minimum of table $\mathcal{T}_{uw}$ and table $\mathcal{T}_{wv}$, is very much similar to the algorithm of composing two tables. So, it takes $O(\log n)$ time with $\frac{n^2}{\log n}$ processors on a CREW PRAM. For finding minimum of n tables we can proceed in a manner analogous to finding minimum of n numbers. Thus, the lemma follows from the fact that minimization operation requires to find the minimum table of n tables. ∎

Lemma 8 [21] *After the first* k *iterations of Step 3 of the pointer doubling algorithm, the correct value of* $\mathcal{T}_{u,v}$ *has been computed for every pair of nodes* u *and* v *when only routes of length at most* 2^k *are taken into account*

Theorem 2 *All pair S-T routing problem can be solved in* $O(\log^3 n)$ *time using* $\frac{n^4}{\log^2 n}$ *processors on a CREW PRAM.*

Proof: The correctness of the proof follows from Lemma 8. Steps 1 and 2 can be performed in $O(1)$ time using at most $O(n^2)$ processors. Therefore we need to show that Step 3 can be performed in $O(\log^2 n)$ time using $\frac{n^4}{\log^2 n}$ processors. In particular, Step 3 involves considering all pairs u and v in parallel, it suffices to show that Steps 3(a)-3(d) can be performed in $O(\log^2 n)$ time with $\frac{n^2}{\log^2 n}$ processors.

For any pair (u, v) in Step 3(a), we independently check all nodes w to see if they are candidates for A; this takes $O(1)$ time with n processors. Since Step 3(b) can be performed in $O(\log n)$ time with $\frac{n^2}{\log n}$ processors by Lemma 6, it can also be performed in $O(\log^2 n)$ time with $\frac{n^2}{\log^2 n}$ processors (Since CREW PRAM is a self-simulating model). Step 3(c) can be performed in $O(\log^2 n)$ time with $\frac{n^2}{\log^2 n}$ processors by Lemma 7. Finally Step 3(d) takes $O(1)$ time using one processor. ∎

let $C(i,j;L)$ (respectively $C(i,j;R)$) be the cost of uniform route on network induced by the subgraph $a_1,\ldots,a_1,D,b_1,\ldots,b_j$ in which a_i is the last element of the sequence (b_j is the last element respectively). The initial conditions for the dynamic programming algorithm are $C(0,0;L) = C(0,0;R) = 0$.
Then, for $i,j > 0$,

$$C(i,j;L) = \min\{C(i-1,j;L) + t(a_{i-1},a_i), C(i-1,j;R) + t(b_j,a_i)\} \quad (1)$$

$$C(i,j;R) = \min\{C(i,j-1;R) + t(b_{j-1},b_j), C(i,j-1;L) + t(a_j,b_i)\} \quad (2)$$

Theorem 3 *[41] dSVRPTW-line problem can be solved sequentially in $O(n^2)$ time where n is the number of locations.*

Proof: The algorithm proceeds by computing $C(i,j,L)$ and $C(i,j,R)$ for $0 \le i \le m$ and $0 \le j \le p$. Each value of C can be computed in constant time using above given equations. Since there are $O(n^2)$ values to be computed, the theorem follows. ∎

3.4.2 Parallel algorithms for dSVRPTW-line problem

Our parallel algorithms are based on the dynamic programming formulation in Theorem 3. We construct a configuration network for a given instance graph G with nodes representing uniform routes in G and edges representing extensions of one route to another by the addition of one new node. Interestingly the configuration network formed is a LDAG. We then use the variants of shortest path algorithms for LDAG to find the shortest route in configuration network.

3.4.2.1 Layered graph construction

For $G = a_m,\ldots,a_1,D,b_1,\ldots,b_p$, the *configuration network* is a layered directed acyclic graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$, where

1. $n = m + p + 1$
2. $\mathcal{V} = \{(i,j,R),(i,j,L) : 1 \le i \le m, 1 \le j \le p\} \cup \{(0,0,D)\} \cup \{(m+1,p+1,D)\}$;
3. The source and sink node between which we need to find the shortest path are $S(\mathcal{G}) = (0,0,D)$ and $T(\mathcal{G}) = (m+1,p+1,D)$ respectively;
4. All the nodes have zero handling time, the dead line of $(0,0,D)$ is 0 and the deadline of $(m+1,p+1)$ is ∞. The deadlines of $(i,j,L),(i,j,R)$ are $d(a_i)$ and

$d(b_j)$ respectively;

5. there are edges from (m, p, L) and (m, p, R) to $(m+1, p+1, D)$ with travel time as 0;
6. there are edges from $(0, 0, D)$ to $(1, 0, L)$ and $(0, 1, R)$ with travel time as $t(a_1, D)$ and $t(D, b_1)$ respectively;
7. for $1 \leq i \leq m$ and $1 \leq j \leq p$ there are edges
   from (i, j, L) to $(i+1, j, L)$ with travel time $t(a_i, a_{i+1})$;
   from (i, j, L) to $(i, j+1, R)$ with travel time $t(a_i, b_{j+1})$;
   from (i, j, R) to $(i, j+1, R)$ with travel time $t(b_j, b_{j+1})$;
   from (i, j, R) to $(i+1, j, L)$ with travel time $t(b_j, a_{i+1})$;
8. there are $n+1$ layers in the *configuration network* $\mathcal{G}$;
9. $(0, 0, D)$ will be in the first layer, and $(m+1, p+1)$ in last layer respectively. Further node (i, j, L), (i, j, R) will be in $(i+j)^{th}$ layer;
10. $b = \min(m, n-m)$;
11. maximum number of nodes in any layer will be $2b+1$
12. the number of nodes in any layer k will be $\begin{cases} 2(i-1) & 2 \leq i \leq b+1 \\ 2b+1 & b+2 \leq i \leq n-b+1 \\ 2(n-i+2) & n-b+2 \leq i \leq n+1 \end{cases}$
13. first layer will have only source node and last layer will have sink node;
14. a cost vector or cost table $\mathcal{T}_{u,v}$ is associated with each $(u, v) \in \mathcal{E}$;

The node (i, j, L) denotes the optimal route covering $[a_i, b_j]$ with a_i being the last node visited. Similarly the node (i, j, R) denotes the optimal route covering $[a_i, b_j]$ with b_j being the last node visited. Node $(0, 0, D)$ and $(m+1, p+1, D)$ denote the null route and optimal route respectively.

Lemma 10 [21] *For $\mathcal{G}$ the configuration network of a line G, the optimal $S(\mathcal{G})$-$T(\mathcal{G})$ route corresponds to an optimal uniform covering route of G.*

Proof: From the construction of $\mathcal{G}$ and Theorem 3, it follows that a route from $S(\mathcal{G})$ to $T(\mathcal{G})$ in $\mathcal{G}$ with cost C corresponds to a uniform route in G whose cost is also C. As well, given an optimal uniform route $\mathcal{P}$ in G of cost C there is a route in $\mathcal{G}$ from $S(\mathcal{G})$ to $T(\mathcal{G})$ of cost C since every initial segment of $\mathcal{P}$ is also a uniform route and the nodes of $\mathcal{G}$ describe all uniform routes as well as transitions from one uniform route to another in one step. Thus, the lemma follows. ∎

3.4.2.2 $O(n)$ time Optimal parallel algorithm

In this section we describe $O(n)$ optimal parallel algorithm for dSVRPTW-line problem, when the vehicle starts at time 0

1. Construct a LDAG for a given instance of the problem as described above in Section 3.4.2.1
2. for $n+1$ iterations do
   begin
   remove all nodes at second layer;
   end;

For removing nodes at second layer, we allocate one processor to every node in third layer and find the shortest path from source node to that node. Since the in-degree of any node in LDAG is 2, the number of paths from source node to any node in third layer is 2. So, we can find the shortest path from source node to any node in third layer in $O(1)$ time. Since there can be at most $n+1$ nodes in third layer in any iteration (the maximum number of nodes in any layer before first iteration is $n+1$). we need n processors in each iteration. Since there are $n+1$ iterations it we takes $O(n)$ time. Thus, we can find an optimal schedule for coalescing operations when number of jobs is two in $O(n)$ time with $O(n^2)$ work.

3.4.2.3 $O(\log^2 n)$ time Parallel algorithm

This algorithm is much faster than the $O(n)$ time optimal algorithm given above. The algorithm computes the optimal path for all vehicle starting times in interval $[0, \infty)$.

1. Construct a LDAG for a given instance of the problem as described above in Section 3.4.2.1
2. for $t = 0$ to $\log n + 1$ do /* $O(\log n)$ iterations */
   for $i = 2$ to $\frac{n+1}{2^t} - 1$ do
   parbegin
   if i is even then remove nodes at layer i;
   parend;

The algorithm for removing all nodes at layer i is given below. Here, d is the maximum in-degree (or out-degree) of any node in the current iteration. Clearly, $d \leq n+1$.

for each node r of layer $i+1$ do /* at most $2*b+1$ times */
for each node v at this layer(i) incident from r do /* at most d times */
for each node p of layer $i-1$ incident from v do /* at most d times */
parbegin
(a) $\mathcal{T}^v_{r,p} = compose(\mathcal{T}_{r,v}, \mathcal{T}_{v,p})$
(b) $\mathcal{T}_{r,p} = \min_v(\mathcal{T}^v_{r,p})$
parend;

3.4.2.4 Analysis of Algorithm

Lemma 11 *The maximum degree of any node before k^{th} iteration will be* $\min(2^k, 2b+1)$

Proof: Any node in the graph has degree less than or equal to the degree of source node of the shortest path to be found. This is because, the source node satisfies the same constraints as any other nodes in the graph. So, it is enough if we prove that the degree of source is $\min(2^k, 2b+1)$ before k^{th} iteration. It is evident from algorithm that after every iteration the number of layers decreases by a factor of 2; further after any iteration layer i will become layer $\left(\frac{i+1}{2}\right)$ (if i is odd). Therefore, in k^{th} iteration the layer adjacent to first layer (second layer) will be same as the layer $2^{k-1}+1$ before the first iteration. (This can be proved by simple induction). The number of nodes in layer i before first iteration is $2(i-1)$. Therefore, the number nodes in second layer before k iterations will be $2(2^{k-1}) = 2^k$. It is evident from description of our graph that every node is reachable from the source (this implies that every node in the second layer is connected to source) and the maximum number of nodes in any layer is at most $2b+1$ so, the degree of a node in any iteration can not exceed $2n+1$. Therefore, the degree of source node is $\min(2^k, 2b+1)$ before k^{th} iteration. Thus, the result follows. ∎

Lemma 12 *For removing nodes at layer i in k^{th} iteration, the algorithm on a CREW PRAM takes $O\left(\log(\min(2b+1, 2^k) * (\log n))\right)$ time using $\left(\frac{bn\min(2^2, 2^{2k})}{\log m * \min(k, \log b)}\right)$ processors.*

Proof: For removing nodes at layer i we need to find $\mathcal{T}_{r,p}$ where, $r \in$ layer $i-1$ and $p \in$ layer $i+1$. For finding $\mathcal{T}_{r,p}$ we need to find minimum of $\mathcal{T}^v_{r,p}$ where, $v \in$ layer i and r

is incident on r, and v is incident on p. If d is the degree of r then we need to compose d tables and find minimum among these d tables. By Lemma 6 and 7, we can compose two tables or find minimum of two tables in $O(\log n)$ time with $\frac{n}{\log n}$ processors. We can compose d pairs of tables into d tables in $O(\log d * \log n)$ time with $\frac{d*n}{\log d*\log n}$ processors. We can also find minimum of d tables in $O(\log d * \log n)$ time with $\frac{d*n}{\log d*\log n}$ processors. Therefore, we can find $\mathcal{T}_{r,p}$ in $O(\log d * \log n)$ time with $\frac{d*n}{\log d*\log d}$ processors on a CREW PRAM. Since there can be d nodes in layer $i+1$ associated with v, and as there can be at most $(2b+1)$ nodes in layer $i-1$ the number of processors for removing nodes at layer i is $(2b+1)*d*\frac{d}{\log d}*\frac{n}{\log n}$. We have by Lemma 11 that the degree of any node in k^{th} iteration is $\min(2b+1, 2^k)$. Therefore, we need $O(2b+1)*\left(\frac{\min(2b+1,2^k)^2}{\log\min(2b+1,2^k)}*\frac{n}{\log n}\right)$ processors and $O\left(\log(\min(2b+1,2^k)*\log n)\right)$ time for k^{th} iteration. ∎

Theorem 4 *There is a CREW PRAM algorithm for dSVRPTW-line that takes* $O(\log^2 n * \log b)$ *time using* $\frac{n^2b^2}{\log b*\log n}$ *processors.*

Proof: The initial number of layers in the graph is $n+1$. In every iteration we will be removing half of the layers. Therefore, there will be $\left(\frac{n+1}{2^{k-1}}\right)$ layers before k^{th} iteration. Combining this with Lemma 12, if in k^{th} iteration we use $\left(\left(\frac{n+1}{2^{k-1}}\right)*(2b+1)*\left(\frac{\min(2b+1,2^k)^2*n}{\log\min(2b+1,2^k)*\log n}\right)\right)$ processors, time will be $O\left(\log\min(2b+1,2^k)\right)$. These expressions will be maximum when $2^{k-1} \geq 2b+1$ or when $k \geq \log(2b+1)+1$. Therefore, the maximum number of processors in any iteration is $\frac{n^2b^2}{\log b*\log n}$. Since each iteration takes at most $O(\log n * \log b)$ time and there are $\log n + 1$ iterations, the time required for our algorithm is $O(\log^2 n * \log b)$. Thus the result follows. ∎

3.5 Traveling Repairman Problem

Afrati *et al.* [1] have described an $O(n^2)$ algorithm for special case of traveling repairman problem when handling time is zero and all the locations are in a line without having time windows associated with them (See Tsitsiklis [41]). But, because of non-availability of that paper we describe our own $O(n^2)$ time dynamic programming algorithm for this problem in Section 3.5.1. In Section 3.5.2 we give parallel algorithms for this special case. Our parallel algorithm makes use of the dynamic programming formulation combined with shortest path algorithm for LDAG.

Let G be an instance of TRP-line. We can fix G to be the network ordered in the

following canonical form $a_m, a_{m-1}, \ldots, a_1, D, b_1, \ldots, b_p$; where D is the central depot from where the vehicle originates and terminates. Since all the locations are on a line there are edges between two consecutive locations only. Every edge will have cost associated with it denoting the time taken for the vehicle to travel between the two locations. The optimal feasible routes for G can be assumed to be in a certain canonical form. For ease of exposition, we can assume that the nodes a_0 and b_0, both refer to the depot D.

Definition 20 *Let $S = v_1, \ldots, v_{m+p+1}$ be an optimal feasible route that covers G. Then S is* uniform *if for any $k \leq m+p+1$, $v_1, \ldots, v_k$ is an optimal feasible route that covers the subgraph of G induced by $a_i, a_{i-1}, \ldots, a_1, D, b_1, \ldots, b_j$ for some i and j such that $k = i+j+1$.*

3.5.1 Sequential algorithm

Let $v_1, \ldots, v_{m+p+1}$ be a uniform route in G. We can represent the subsequence $v_1, \ldots, v_k$ by the ordered pair (i, j) where a_i and b_j both appear in this subsequence and $i+j+1 = k$. Then, the subsequence $v_1, \ldots, v_k, v_{k+1}$ is represented by either $(i+1, j)$ or $(i, j+1)$ depending on whether v_{k-1} is a_{i+1} or b_{j+1}. More generally, for $0 \leq i \leq m$ and $0 \leq j \leq p$, let $C(i, j; L)$ and $C(i, j; R)$ be the cost of uniform route on network induced by the subgraph $a_i, \ldots, a_1, D, b_1, \ldots, b_j$ in which a_i is the last element of the sequence (b_j is the last element respectively) and let $T(i, j; L)$, $T(i, j; R)$ be the times taken for these uniform routes. The initial conditions for the dynamic programming algorithm are $C(0, 0; L) = C(0, 0, R) = T(0, 0; L) = T(0, 0; R) = 0$.

Then, for $i, j > 0$,

$$\text{Let } \alpha = T(i, j-1; L) + t(i, j), \tag{3}$$

$$\beta = T(i, j-1; R) + t(j-1, j), \tag{4}$$

$$\gamma = T(i+1, j; L) + t(i, i+1) \text{ and} \tag{5}$$

$$\delta = T(i+1, j; R) + t(i, j). \tag{6}$$

$$\text{Then } C(i, j, R) = \min\{C(i, j-1, L) + \alpha,\ C(i, j-1, R) + \beta\} \tag{7}$$

$$C(i, j, L) = \min\{C(i+1, j, R) + \gamma,\ C(i+1, j, L) + \delta\} \tag{8}$$

$$C(i,j,R) = \begin{cases} \alpha & if\ C(i,j,R) = C(i,j-1:L)+\alpha \\ \beta & if\ C(i,j:R) = C(i,j-1:R)+\beta \end{cases} \quad (9)$$

$$C(i,j,L) = \begin{cases} \gamma & if\ C(i,j:L) = C(i+1,j:R)+\gamma \\ \delta & if\ C(i,j;L) = C(i+1,j;L)+\delta \end{cases} \quad (10)$$

(11)

Theorem 5 *The special case of TRP-line in which the handling times of all nodes is zero can be solved in $O(n^2)$ where n is the number of locations.*

Proof: The algorithm proceeds by computing $C(i,j,L)$ and $C(i,j,R)$ for $0 \leq i \leq m$ and $0 \leq j \leq p$. Each value of C can be computed in constant time using above given equations. Since there are $O(n^2)$ values to be computed, the theorem follows ∎

3.5.2 Parallel algorithms for TRP-line problem

Our parallel algorithm is based on the dynamic programming formulation in Theorem 5. We construct a configuration network for a given instance graph G with nodes representing uniform routes in G and edges representing extensions of one route to another by the addition of one new node. Interestingly the configuration network formed is a LDAG. We then use the variants of shortest path algorithms for LDAG to find the shortest route in configuration network.

3.5.2.1 Layered graph construction

We can construct a layered graph that is similar to one in the dSVRPTW-line problem. There will be $n-1$ layers as in the dSVRPTW-line problem. For $G = a_m, \ldots, a_1, D, b_1, \ldots, b_p$, the *configuration network* is a layered directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. where

1. $n = m + p + 1$
2. $\mathcal{V} = \{(i,j,R),(i,j,L) : 1 \leq i \leq m, 1 \leq p\} \cup \{(0,0,D)\} \cup \{(m+1,p+1,D)\}$;
3. The source and sink node between which we need to find the shortest path are $S(\mathcal{G}) = (0,0,D)$ and $T(\mathcal{G}) = (m+1,p+1,D)$ respectively;
4. If (u,v) edge is in the graph then it denotes optimal cost route between the vertices u and v. We associate three values $d(u,v)$, $a(u,v)$ and $len(u,v)$ with each edge. let $u, u_1, u_2, \ldots, u_l, v$ be the optimal cost route between the vertices u and v then $d(u,v)$ denotes the sum of waiting times of nodes in that path

(i.e. $d(u,v) = (l+1) * t(u,u_1) + l * t(u_1,u_2) + \ldots + 2 * t(u_{l-1},u_l) + t(u_l,v)$), $a(u,v)$ denotes the traveling time from u to v in the optimal path (i.e. $a(u,v) = t(u,u_1) + t(u_1,u_2) + t(u_2,u_3) + \ldots + t(u_{l-1},u_l) + t(u_l,v)$) and $len(u,v)$ denotes the length of the optimal path (i.e. $l+1$). The initial values of $d(u,v)$, $a(u,v)$ are $t(u,v)$. Where $t(u,v)$ denotes the travel time between those nodes. The initial values of $len(u,v)$ is 1 if $t(u,v) \neq 0$, else $len(u,v)$ is 0.

5. there are edges from (m,p,L) and (m,p,R) to $(m+1,p+1,D)$.
6. there are edges from $(0,0,D)$ to $(1,0:L)$ and $(0,1,R)$.
7. for $1 \leq i \leq m$ and $1 \leq j \leq p$ there are edges from (i,j,L) to $(i+1,j,L)$, from (i,j,L) to $(i,j+1,R)$ from (i,j,R) to $(i,j+1,R)$, from (i,j,R) to $(i+1,j,L)$
8. source and sink will be in first layer and last layer respectively. (i,j,L), (i,j,R) will be in $(i+j)^{th}$ layer;
9. $b = \min(m, n-m)$;
10. maximum number of nodes at any layer will be $2b+1$

The node (i,j,L) denotes the optimal route covering $[a_i,b_j]$ with a_i being the last node visited. Similarly the node (i,j,R) denotes the optimal route covering $[a_i,b_j]$ with b_j being the last node visited. Node $(0,0,D)$ and $(m+1,p+1,D)$ denote the null route and optimal route respectively.

3.5.2.2 $O(n)$ time Optimal parallel algorithm

1. Construct a LDAG for a given instance of the problem as described above in Section 3.5.2.1
2. for $n+1$ iterations do
   begin
   remove all nodes at second layer:
   end;

For removing nodes at second layer, we allocate one processor to every node in third layer, and find the a from source node to that node such that sum of waiting times of nodes in path is minimum. Since the in-degree of any node in LDAG is 2, the number of paths from source node to any node in third layer is 2. So, we can find the shortest path

path from source node to any node in third layer in $O(1)$ time. Since there can be at most $n+1$ nodes in third layer in any iteration (the maximum number of nodes in any layer before first iteration is $n+1$). we need n processors in each iteration. Since there are $n+1$ iterations it takes $O(n)$ time. Thus, the special case of TRP-line in which the handling times of all nodes is zero can be solved in $O(n)$ time with $O(n^2)$ work.

3.5.2.3 $O(\log^2 n)$ time parallel algorithm for TRP-line problem

1. Construct a LDAG for a given instance of the problem as described above in Section 3.5.2.1.
2. for $t = 0$ to $\log n + 1$ do /* $O(\log n)$ iterations */
   for $i = 2$ to $\frac{n+1}{2^t} - 1$ do
   parbegin
   if i is even then remove nodes at layer i;
   parend;

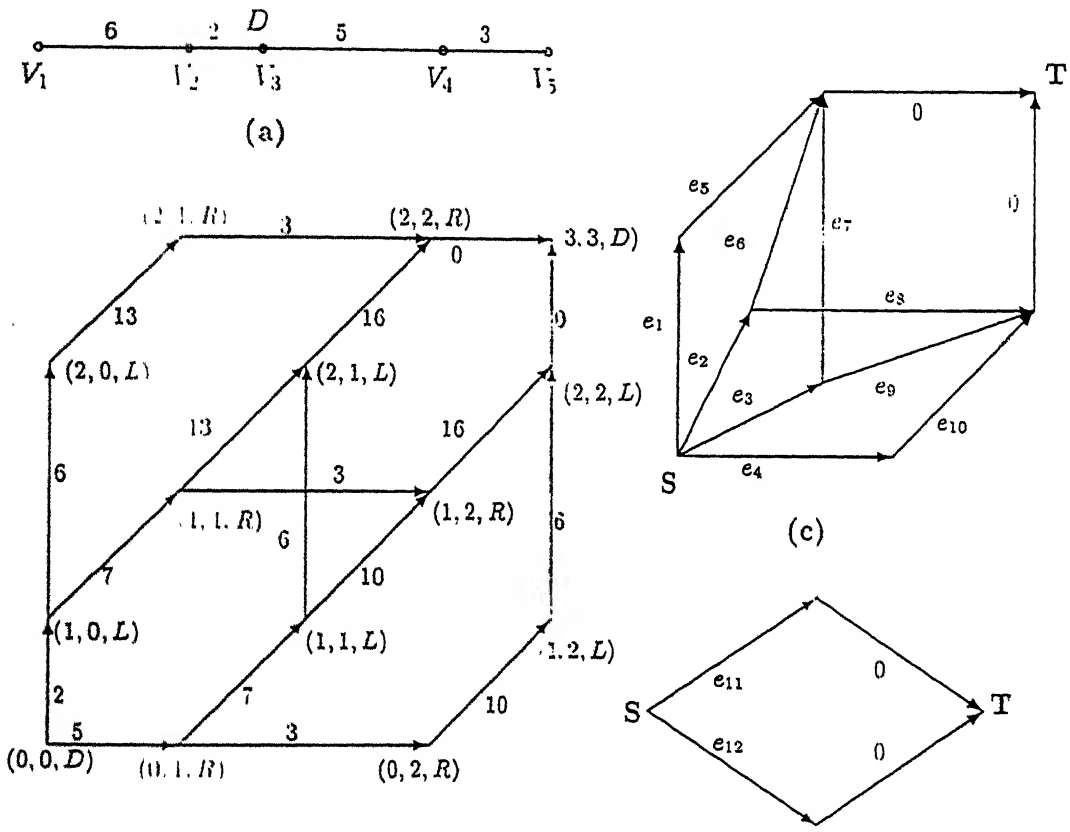
The algorithm for removing all nodes at layer i is given below. Here, d is the maximum in-degree (or out-degree) of any node in the current iteration. Clearly, $d \leq 2n+1$.

for each node r of layer $i-1$ do /* at most $2b+1$ times */
for each node v at this layer i incident from r do /* at most d times */
for each node p of layer $i+1$ incident from v do /* at most d times */
parbegin
(a) $d(r,p) = \min_v(d(r,v) + d(v,p) + len(v,p) * a(r,v))$;
(b) Let u be the node that gives rise to the minimum value of $d(r,p)$;
(c) $a(r,p) = a(r,u) + a(u,p)$;
(d) $len(r,p) = len(r,u) + len(u,p)$;
parend;

Observe that the algorithm is similar to dSVRPTW-line problem. The correctness proof will also be similar. It can be easily proven that above algorithm takes $O(\log n * \log b)$ time with $O(nb^2/\log b)$ processors on a CREW PRAM where $b = \min(m, n-m)$.

3.5.2.4 An example

Figure 3(a) shows a TRP-line with five nodes $V_1 \ldots V_5$. Figure 3(b) shows the LDAG constructed following the procedure given in above in the section. It has six layers. We have shown only travel times between the nodes for simplicity. The reader can easily infer $d(u,v), a(u,v)$ and $len(u,v)$ associated with edges. Figure 3(c) and Figure 3(d) shows the LDAG in second and third iterations respectively. Optimal solution path for



	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}
$d(e)$	10	11	17	13	29	42	28	22	36	26	55	51
$a(e)$	8	9	12	8	16	29	22	19	26	16	24	28
$l(e)$	2	2	2	2	2	2	2	2	2	2	4	4

Figure 3: An example describing our parallel algorithm for TRP-line problem

this example is D, V_2, V_4, V_5, V_1, the sum of waiting times of all nodes is 51 and the cost of the path associated with the optimal solutions is 28.

Chapter 4

Parallel Algorithms For Coalescing Operations with Precedence Constraints In Real-time Systems

4.1 Introduction

In many applications [7], real-time systems are modeled as object-oriented systems. In an object-oriented system, each object has a set of well defined operations. Each object also has local variables, which may be accessed only by the operations defined in the interface of the object. The object decides if and when to process the requests from other objects.

In most applications, coalesced operations must be defined by the programmer, due to the semantic issues involved. On the other hand, whether a coalescence should be performed or not is usually decided by the object (or system) scheduler. In the scheduler, the *reward* (usually the amount of computation time saved) for each coalesced operation must be pre-analyzed and recorded in a *reward table* before execution. When a sequence of requests arrives, the scheduler consults the reward table and determines which requests to coalesce in order to maximize the total reward.

In a real-time system, the request patterns of many jobs are known in advance, especially when jobs are periodic. For example, service requests from a radar monitor are periodic and well-defined. Given K periodic jobs and the reward table, we can

determine which requests to coalesce to produce the maximum total reward. In other words, we can find the coalescence schedule that saves the most time. If some requests are not predefined, we can still perform the analysis at run-time if the complexity of scheduling coalesced operations is not high. Bihari *et al.* [7] have proved that finding an optimal schedule in the general case is $\mathcal{NP}$-hard.

For the case of two periodic jobs, Chen *et al.* [9] have developed an $O(n^6)$ time sequential algorithm and Liu *et al.* [30] have reduced it to $O(n^2)$ time. We show that there is an an $O(\log^2 n)$ time parallel algorithm using $O(n^3/\log n)$ processors for this problem on a CREW PRAM.

4.2 Scheduling two periodic jobs

Liu *et al.* [30] have used dynamic programming to obtain a quadratic time algorithm for finding optimal schedule when the number of jobs is two. Our algorithm makes use of the parallel algorithm for the shortest path problem for layered directed acyclic graph (LDAG) combined with the dynamic programming formulation of Liu *et al.* We describe the sequential algorithm proposed by Liu *et al.* [30] in Section 4.2.1. We give an example for the problem in Section 4.2.2. In Section 4.3 we will use this dynamic programming formulation for constructing our parallel algorithm.

4.2.1 Sequential algorithm

Each job J_i has n sequential operations $J_{i,1}, J_{i,2}, \ldots, J_{i,n}, i = 1, 2$. The system can be represented as an undirected graph. Each vertex $v_{i,j}$ denotes an operation $J_{i,j}$. There are edges between $v_{i,j}$ and $v_{i,j+1}$. $i = 1, 2$, $j = 1, 2, \ldots, n-1$ and between $v_{1,p}$ and $v_{2,q}$, $p = 1, 2, \ldots, n$, $q = 1, 2, \ldots, n$. Each edge is associated with a weight representing the reward value. Thus, maximizing the total reward is equivalent to finding a *maximum weighted compatible matching* in the graph [30]. Any two edges $(v_{1,r}, v_{2,s})$, $(v_{1,p}, v_{2,q})$ in a weighted compatible matching satisfy either $r < p$ and $s < q$ or $r > p$ and $s > q$. In other words, the maximum weighted compatible matching problem is to find a subset of edges that do not cross each other and whose total weight is maximized. This subset of edges is called the *maximum weighted compatible matching set* [7].

Let $R_{i,j}$ denote the maximum total reward for matching operations in $J_{1,1}, J_{1,2}, \ldots, J_{1,i}$,

and $J_{2,1}, J_{2,2}, \ldots, J_{2,j}$, $r_{i,j}$ the weight of edge $(v_{1,i}, v_{2,j})$, $1 \leq i \leq n$, $i \leq j \leq n$, and $c_{k,l}$ the weight of edge $(v_{k,l-1}, v_{k,l})$, $k = 1, 2, 1 < l \leq n$. Clearly, considering operations $J_{1,i}$ and $J_{2,j}$, we have the following equations[30]:

$$
\begin{aligned}
R_{i,j} &= \max\{R_{i-1,j},\ R_{i,j-1},\ c_{1,i} + R_{i-2,j},\ c_{2,j} + R_{i,j-2},\ r_{i,j} + R_{i-1,j-1}\} \\
&\quad for\ 2 \leq i \leq n,\ 2 \leq j \leq n, && (12)\\
R_{1,j} &= \max\{R_{0,j},\ R_{1,j-1},\ c_{2,j} + R_{i,j-2},\ r_{1,j} + R_{0,j-1}\}\ \ for\ 2 \leq j \leq n, && (13)\\
R_{i,1} &= \max\{R_{i-1,1},\ R_{i,0},\ c_{1,i} + R_{i-2,j},\ r_{i,1} + R_{i-1,0}\}\ \ for\ 2 \leq i \leq n, && (14)\\
R_{0,j} &= \max\{R_{0,j-1},\ c_{2,j} + R_{i,j-2}\}\ \ for\ 2 \leq j \leq n, && (15)\\
R_{i,0} &= \max\{R_{i-1,0},\ R_{i,0},\ c_{1,i} + R_{i-2,j}\}\ \ for\ 2 \leq i \leq n, && (16)\\
R_{0,0} &= R_{1,0} = R_{0,1} = 0, \\
R_{1,1} &= r_{1,1}\ \ if\ \ r_{1,1} > 0,\ and\ R_{1,1} = 0\ \ otherwise. && (17)
\end{aligned}
$$

In Equation 12, the first (second) term in the right-hand side represents the case that operation $J_{1,i}$ ($J_{2,j}$) is not coalesced with any other operation. The third (fourth) term represents the case that operation $J_{1,i}$ ($J_{2,j}$) is coalesced with its immediate predecessor. The fifth term represents the case that $J_{1,i}$ is coalesced with $J_{2,j}$. No other cases are possible, because edges in the maximum weighted compatible matching set are not allowed to cross each other due to precedence constraints. The other equations are similarly derived. With initial values given by Equation 17, we can compute the value of $R_{n,n}$, which is the maximum total reward in $O(n^2)$ time.

4.2.2 An example

	op_1	op_2	op_3	op_4
op_1	2	7	5	0
op_2	7	2	0	4
op_3	5	0	2	1
op_4	0	4	1	2

Table 3: Reward table of the example

Figure 4 shows a real-time object with four primitive operations op_1, op_2,op_3, and

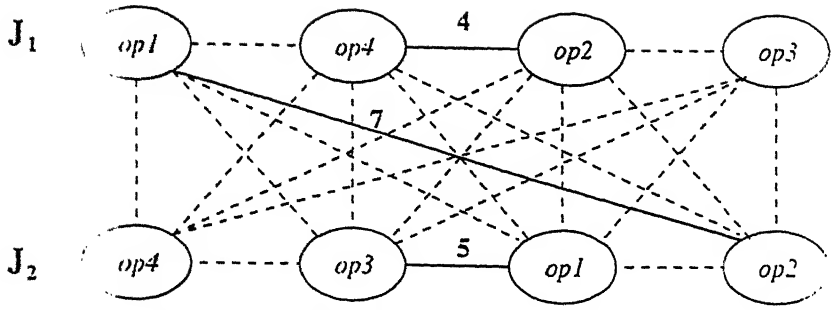


Figure 4: An example of scheduling operations with precedence constraints

op_4. There are two periodic jobs and each has four operations. Job J_1 consists of op_1, op_4,op_2, and op_3 ($J_{1,1}$, $J_{1,2}$, $J_{1,3}$, $J_{1,4}$ respectively). Job J_2 consists of op_4, op_3, op_1, and op_2 ($J_{2,1}$, $J_{2,2}$, $J_{2,3}$, $J_{2,4}$, respectively). The reward values are given in Table 3. The maximum total reward is 16 and the scheduling sequence is $J_{2,1}$, $J_{2,2} + J_{2,3}$, $J_{1,1} + J_{2,4}$, $J_{1,2} + J_{1,3}$, $J_{1,4}$.

4.3 Parallel algorithms for scheduling two jobs

Our parallel algorithms are based on dynamic programming formulation presented in Section 4.2.1. We construct a layered graph for a given instance of problem. We then use the algorithm described in Section 2.3 for finding the optimum schedule.

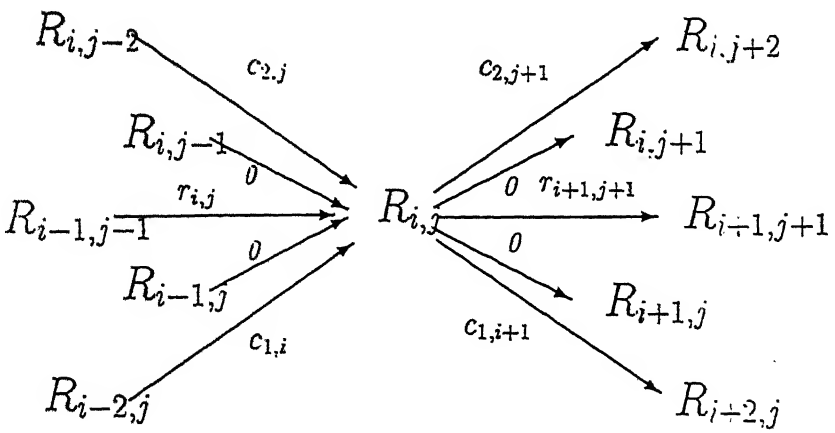


Figure 5: Information flow in equations given in Section 4.2.1

Figure 5 shows the information flow of the equations given in Section 4.2.1. If we try to construct a graph by placing R_{ij} in $(i+j)^{th}$ layer (say l) there will be edges

between layer l and $l+2$. This will not be a layered graph. In order to construct a LDAG, we regroup the terms in Equations given in Section 4.2.1. In Equation 12 $R_{i,j}$ was calculated as maximum of five terms. If we place $R_{i,j}$ in layer $(i+j)^{th}$ (say l) layer then $R_{i,j-2}$, $R_{i-1,j-1}$ and $R_{i-2,j}$ will be in layer $l-2$. So, we find the maximum of these three terms and put it in layer $l-1$ and then find the maximum among remaining two terms ($R_{i-1,j}$, $R_{i,j-1}$) and new term that we put in layer $l-1$. Then, we will have edges from l to $l+1$ only. Thus forming a LDAG. For ease of representation we replace $R_{i,j}$ with $R'_{2(i+1),2(j+1)}$. Thus we get following Equations 18 and 19. Similarly, we get Equations 20 and 21 from Equation 13, Equations 22 and 23 from Equation 14, Equations 24 and 25 from Equation 15, Equations 26 and 27 from Equation 16. Thus we can rewrite the equations as follows:

$$
\begin{aligned}
R'_{2i,2j} &= \max\{R'_{2i-2,2j},\ R'_{2i,2j-2},\ R'_{2i-1,2j-1}\} \\
&\quad for\ 3 \le i \le n+1,\ 3 \le j \le n+1, && (18)\\
R'_{2i-1,2j-1} &= \max\{c_{1,i-1} + R'_{2i-4,2j},\ c_{2,j-1} + R'_{2i,2j-4},\ r_{i-1,j-1} + R'_{2i-2,2j-2}\} \\
&\quad for\ 3 \le i \le n+1,\ 3 \le j \le n+1, && (19)\\
R'_{4,2j} &= \max\{R'_{2,2j},\ R'_{4,2j-2},\ R'_{3,2j-1}\} \\
&\quad for\ 3 \le j \le n+1, && (20)\\
R'_{3,2j-1} &= \max\{c_{2,j-1} + R'_{4,2j-4},\ r_{1,j-1} + R'_{2,2j-2}\} \\
&\quad for\ 3 \le j \le n+1, && (21)\\
R'_{2i,4} &= \max\{R'_{2i-2,2},\ R'_{2j-2,4},\ R'_{2i-1,3}\} \\
&\quad for\ 3 \le i \le n+1, && (22)\\
R'_{2i-1,3} &= \max\{c_{1,i-1} + R'_{2i-4,4},\ r_{i-1,1} + R'_{2i-2,2}\} \\
&\quad for\ 3 \le i \le n+1, && (23)\\
R'_{2,2j} &= \max\{R'_{2,2j-2},\ R'_{1,2j-1}\} \quad for\ 3 \le j \le n+1, && (24)\\
R'_{1,2j-1} &= c_{2,j-1} + R'_{2,2j-4} \quad for\ 3 \le j \le n+1, && (25)\\
R'_{2i,2} &= \max\{R'_{2i-2,2},\ R'_{2i-1,1}\} \quad for\ 3 \le i \le n+1, && (26)\\
R'_{2i-1,1} &= c_{1,i-1} + R'_{2i-4,2} \quad for\ 3 \le i \le n+1, && (27)\\
R'_{2,2} &= R'_{4,2} = R'_{2,4} = 0,
\end{aligned}
$$

$$R'_{4,4} = r_{1,1} \ if \ r_{1,1} > 0, \ and \ R'_{4,4} = 0 \ \ otherwise. \tag{28}$$

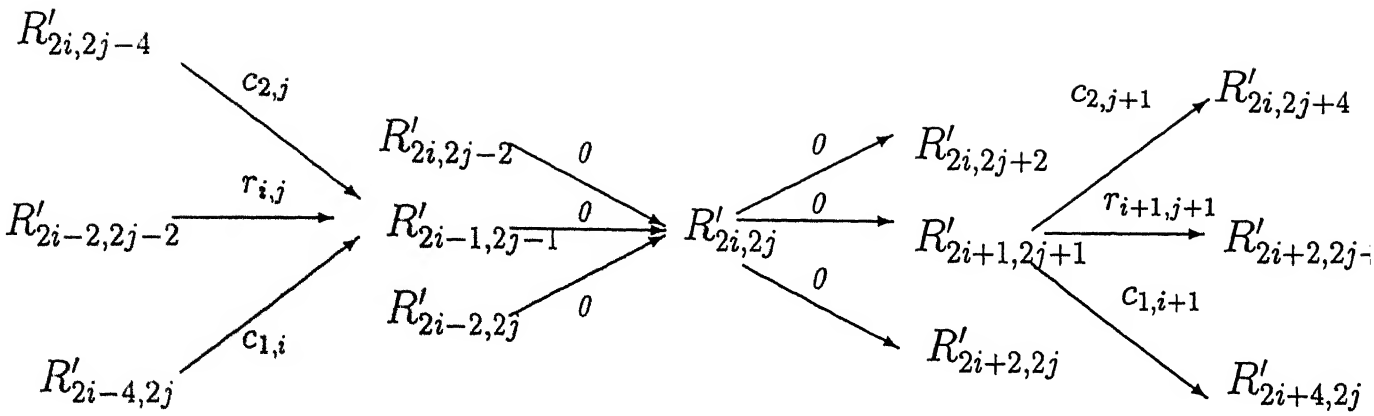


Figure 6: Information Flow in modified equations

Figure 6 shows the information flow in modified equations. In Equation 18, the first (second) term in the right-hand side represents the case that operation $J_{1,i}$ ($J_{2,j}$) is not coalesced with any other operation. In the Equation 19 first (second) term represents the case that operation $J_{1,i}$ ($J_{2,j}$) is coalesced with its immediate predecessor. The third term in Equation 19 represents the case that $J_{1,i}$ is coalesced with $J_{2,j}$. No other cases are possible, because edges in the maximum weighted compatible matching set are not allowed to cross each other due to precedence constraints. The other equations are derived similarly. With initial values given by Equation 28, we can compute the value of $R'_{2n+2,2n+2}$, which is the maximum total reward in $O(n^2)$ time.

4.3.1 Layered graph construction

Given an instance of the problem we can construct a layered graph $G = (V, E)$ where

1. $V = \{(i,j) : 1 \leq i \leq 2(n+1), 1 \leq j \leq 2(n+1)$ where $(i+j)$ is even$\} - \{(1,1),(3,1),(1,3)\}$
2. Let us call the nodes of the form $(2i, 2j)$ as *even nodes* and nodes of the form $(2i+1, 2j+1)$ as *odd nodes.* We can clearly see that nodes other than *even* and *odd* are not possible.

3. The source and sink nodes between which, we have to find the shortest path are $S(G) = (2, 2)$ and $T(G) = (2(n+1), 2(n+1))$ respectively.
4. From Equation 18 we add edges from $(2i-2, 2j)$, $(2i, 2j-2)$ and $(2i-1, 2j-1)$ to $(2i, 2j)$ with costs 0. Similarly for Equation 19 we add the following edges to $(2i-1, 2j-1)$ from $(2i-4, 2j)$ with cost $c_{1,i}$, from $(2i, 2j-4)$ with cost $c_{2,j}$ and from $(2i-1, 2j-1)$ with cost $r_{i,j}$. Thus we will have the following edges for $1 \leq i \leq n+1$ and $1 \leq j \leq n+1$

$$(2i, 2j) \text{ to } \begin{cases} (2i+3, 2j-1) & \textit{with cost } c_{1,i}, \\ (2i+2, 2j) & \textit{with cost } 0. \\ (2i+1, 2j+1) & \textit{with cost } r_{i,j}, \\ (2i, 2j+2) & \textit{with cost } 0, \\ (2i-1, 2j+3) & \textit{with cost } c_{2,j}, \end{cases}$$

   for $1 \leq i \leq n$ and $1 \leq j \leq n$, there is an edge from $(2i+1, 2j+1)$ *to* $(2i+2, 2j+2)$ *with cost* 0
5. number of layers are $2n+1$.
6. node (i, j) will be there in $\frac{i+j}{2} - 1^{th}$ layer.

We observe the following facts from the construction of graph.

Fact 1 *Number of nodes in any layer i is $2(n+1-|n+1-i|)+1$ (at most $2n+1$).*

Fact 2 *Maximum in-degree or out-degree of any node is* 5

4.3.2 $O(n)$ time Optimal parallel algorithm

1. Construct a LDAG for a given instance of the problem as described in Section 4.3.1.
2. for $2n+1$ iterations do

   begin

   remove all nodes at second layer;

   end;

For removing nodes at second layer, we allocate one processor to every node in third layer and find the shortest path from source node to that node. Since the in-degree of any node in LDAG is 3, the number of paths from source node to any node in third layer is 3. So, we can find the shortest path from source node to any node in third layer in $O(1)$ time. Since there can be at most $2n+1$ nodes in third layer in any iteration

(the maximum number of nodes in any layer in first iteration is $2n+1$) we need $O(n)$ processors in each iteration. Since there are $2n+1$ iterations it we will take $O(n)$ time. Thus, we can find an optimal schedule for coalescing operations when number of jobs is two in $O(n)$ time with $O(n^2)$ work.

4.3.3 $O(\log^2 n)$ time parallel algorithm

The parallel algorithm for finding an optimal schedule is as follows:

1. Construct a LDAG for a given instance of the problem as described in Section 4.3.1.
2. for $t = 0$ to $\log(n+1)+1$ do /* $O(\log n)$ iterations */
   for$i = 2$ to $\frac{2n+1}{2^t} - 1$ do
   parbegin
   if i is even then remove nodes at layer i;
   parend;

The algorithm for removing all nodes at layer i is described below. Here, d is the maximum in-degree (or out-degree) of any node in the current iteration. Clearly, $d \leq 2n+1$.

for each node r of layer $i-1$ do /* at most $2n+1$ times */
  for each node v at this layer i incident from r do /* at most d times */
    for each node p of layer $i+1$ incident from v do /* at most d times */
      parbegin
        $d(r,p) = \max_v(d(r,v)+d(v,p))$
      parend;

4.3.3.1 Analysis of Algorithm

Lemma 13 *In k^{th} iteration the layer adjacent to first layer (second layer) will contain all nodes which were present in layer $2^{k-1}+1$ before first iteration.*

Proof: In every iteration we will be removing all even layers. In second iteration second layer is at distance 2 from the old first layer (before the first iteration) i.e. it will be the third layer. After k iterations the second layer will be the layer that was at a distance

2^{k-1} from the old layer 1 (before the first iteration). Thus the lemma follows. ∎

Lemma 14 *At any point in time the degree of any node is less than or equal to the degree of source node of the graph.*

Proof: Since source node is an *even node* its behavior represents the behavior of all *even nodes*. We can observe from the construction of graph that every *odd node* will have only one edge leaving it, and that edge is to an *even node*. So, the degree of *odd node* in k^{th} iteration will be equal to degree of an *even node* in the previous iteration. Thus the lemma follows. ∎

Lemma 15 *The maximum degree of any node after k^{th} iteration will be* $\min(2^k + 3, 2n + 1)$

Proof: By Lemma 14, it follows that it is enough if we prove that the degree of source as $\min(2^k + 3, 2n + 1)$ in k^{th} iteration. By Fact 1, we have the number of nodes in first iteration at layer i is at most $2i+1$ and from Lemma 13 second layer in k^{th} iteration will be layer $2^{k-1} + 1$ in first iteration. Therefore, the number nodes in second layer after k iterations will be $2 * (2^{k-1} + 1) + 1 = 2^k + 3$. It is evident from description of our graph that every node is reachable from the source (this implies that every node in the second layer is connected to source) and the maximum number of nodes in any layer is at most $2n + 1$ so, the degree of a node in any iteration can not exceed $2n + 1$. Therefore, the degree of source node is $\min(2^k + 3, 2n + 1)$ in k^{th} iteration. Thus, the result follows. ∎

Lemma 16 *For removing nodes at layer i in k^{th} iteration, our algorithm on a CREW PRAM requires $O\left(\log(\min(2n + 1, 2^k + 3))\right)$ time using $(2n+1) * \left(\frac{\min(2n+1,2^k+3)^2}{\log \min(2n+1,2^k+3)}\right) = \Theta\left(\frac{n \min(n^2,2^{2k})}{\min(k,\log n)}\right)$ processors.*

Proof: For removing nodes at layer i we need to find $d(r,p)$ where, $r \in$ layer $i-1$ and $p \in$ layer $i+1$. For finding $d(r,p)$ we need to find minimum of $d(r,v) + d(v,p)$ where, v is in layer i and r is incident on v, and v incident on p. If d is the degree of r then we need to find minimum of d numbers, this will require $\frac{d}{\log d}$ processors and $\log d$ time on a CREW PRAM. Since there can be d nodes in layer $i+1$ associated with v and at most $(2n+1)$ nodes in layer $i-1$ the number of processors for removing nodes at layer i is $(2n+1) * d * \frac{d}{\log d}$. We have by Lemma 15 that the degree of any node in k^{th} iteration is $\min(2n + 1, 2^k + 3)$. Therefore, we need $(2n + 1) * \left(\frac{\min(2n+1,2^k+3)^2}{\log \min(2n+1,2^k+3)}\right)$

processors and $O\left(\log(\min(2n+1, 2^k+3))\right)$ time for k^{th} iteration. ∎

Theorem 6 *An optimal schedule for coalescing operations when number of jobs is two can be found in $O(\log^2 n)$ time using $\frac{n^3}{\log n}$ processors on a CREW PRAM.*

Proof: The initial number of layers in the graph is $2n+1$. In every iteration we will be removing half of layers present. Therefore, there will be $\left(\frac{2n+1}{2^k}\right)$ layers in k^{th} iteration. Combining this with Lemma 16, k^{th} iteration takes $\left(\frac{2n+1}{2^k}\right) * (2n+1) * \left(\frac{\min(2n+1,2^k+3)^2}{\log \min(2n+1,2^k+3)}\right)$ processors and $O\left(\log \min(2n+1, 2^k+3)\right)$ time. These expressions will be maximum when $2^k+3 \geq 2n+1$ or when $k \geq \log 2(n-1)$. Therefore, the number of processors in any iteration is $\frac{n^3}{\log n}$. Since each iteration takes at most $O(\log(2n+1))$ time and there are $\log(n+1)+2$ iterations, the time required for our algorithm is $O(\log^2 n)$. Thus the result follows. ∎

It takes $O(\log \log n)$ time to find minimum among n numbers on a common-CRCW PRAM with $\frac{n}{\log \log n}$ processors. An optimal schedule for coalescing operations when number of jobs is two can be solved in $O(\log n \log \log n)$ time using $\frac{n^3}{\log \log n}$ processors on a common-CRCW PRAM.

Chapter 5

Sub-Cubic Cost Algorithm for the All Pairs Longest Path Problem for Directed Acyclic Graph

5.1 Introduction

The all pairs longest path (APLP) problem is to compute longest paths between all pairs of vertices in a directed graph. The all pairs longest distance problem (APLD) is defined similarly, the word "paths" is replaced by "distances". These problems are $\mathcal{NP}$-hard for general graphs. But for directed acyclic graphs, these can be solved in a manner analogous to the all pairs shortest path problem. In this chapter, we modify the all pairs *shortest path* algorithms given by Takaoka [40] to find all pairs longest path for directed acyclic graphs.

In this chapter, we design a parallel algorithm for the APLD problem, for a directed acyclic graph with unit edge costs, with $O(\log^2 n)$ time, (worst case) and $O(n^{(3+w)/2}/\sqrt{\log n})$ processors in Section 5.4.1. Next we describe the parallel algorithm for the APLP problem, with general edge costs in Section 5.4.2. In Section 5.2, we give basic definitions. In Section 5.3, we give an algorithm for longest distance matrix multiplication(LDMM).

5.2 Basic definitions

A directed graph $G(V, E)$, has vertices $V = \{0, 1, \ldots, n-1\}$ and edges E is a subset of $V \times V$. The edge cost $(i, j) \in E$ is denoted by d_{ij}; the matrix D has (i, j) element as d_{ij}. We assume that $d_{ij} = -\infty$ if there is no edge from i to j and $d_{ii} = 0$. The cost, or distance, of a path is sum of costs of the edges in the path. The length of a path is the number of edges in the path. The longest distance from vertex i to vertex j is the maximum cost over all paths from i to j, denoted by d^*_{ij}. Let $D^* = \{d^*_{ij}\}$.

Let A and B are (n, n)-matrices. The three products are defined using the elements of A and B as follows:

$$\text{Ordinary multiplication over a ring } C = AB \quad c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \tag{29}$$

$$\text{Boolean matrix multiplication } C = A \cdot B \quad c_{ij} = \bigvee_{k=0}^{n-1} a_{ik} \wedge b_{kj}, \tag{30}$$

$$\text{Longest distance matrix multiplication} C = A \times B \quad c_{ij} = \max_{0 \le k \le n-1} \{a_{ik} + b_{kj}\} \tag{31}$$

The best known algorithm [13] for ordinary matrix multiplication has time complexity $O(n^\omega)$, $\omega = 2.376$. We can also use that algorithm for Boolean matrix multiplication. If we have an algorithm for LDMM with $T_D(n)$ time, we can solve APLD problem in $O(T_D(n) \log n)$ time by repeated squaring. If we have an algorithm for LDMM with witnesses, (i.e. which represents the set of k which gives the maximum for c_{ij}) with $T_P(n)$ time, we can solve APLP problem in $O(T_P(n) \log n)$ time.

5.3 Longest Distance matrix multiplication by divide-and-conquer

Longest distance matrix multiplication operation can also be viewed as computing n^2 inner products defined, for $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$ by

$$a \times b = \max_{1 \le k \le n} \{a_k + b_k\}$$

Now we divide A, B and C into (m, m)-submatrices for $N = n/m$ as follows.

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & \cdots & B_{1N} \\ \vdots & & \vdots \\ B_{N1} & \cdots & B_{NN} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & \cdots & C_{1N} \\ \vdots & & \vdots \\ C_{N1} & \cdots & C_{NN} \end{bmatrix}$$

Then, C can be computed by

$$C_{ij} = \max_{1 \le k \le N} \{A_{ik} \times B_{kj}\} (i, j = 1, \ldots, N) \tag{32}$$

where the product of submatrices is defined in a similar manner to Equation 31, and the "max" operation is defined on matrices by taking the "max" operation component-wise. Since comparisons and additions of distances are performed in a pair, we can omit counting the number of additions for measurement of complexity. We have to find n^2 maxima of N numbers($O(n^2N)$ time), we have N^3 multiplications of distance matrices in Equation 32. Let us assume that each multiplication of (m, m) matrices can be done in $T(m)$ computing time, assuming that a pre-computed table is available at no cost. The time for constructing the table is given by Lemma 19, and is reasonable when m is small. Then the total computing time is given by

$$O\left(n^2N + N^3T(m)\right) = O\left(n^3/m + (n/m)^3T(m)\right) \tag{33}$$

By Lemma 17 we have, $T(m) = O(m^{2.5})$. Therefore, the time given in Equation 33 becomes $O(n^3/\sqrt{m})$.

5.3.1 Distance matrix multiplication by table-lookup

We divide (m, m)-matrices into strips for $M = m/l$ where $\sqrt{m} \le l \le m$ as follows:

$$A = \left[\boxed{A_1} \ \cdots \ \boxed{A_M} \right] \quad B = \begin{bmatrix} \boxed{B_1} \\ \vdots \\ \boxed{B_M} \end{bmatrix}$$

where A_i is a (m, l) matrix, and B_j is a (l, m) matrix. We regard later A and B as (m, m)-submatrices in Equation 32. Now the product $C = A \times B$ is given by

$$C = \max_{1 \le k \le M} \{A_k \times B_k\} \tag{34}$$

By Lemma 18 we can compute $A_k \times B_k$ in $O(l^2m)$ time, assuming that a pre-computed table is available. Then, the right-hand side of Equation 34 can be computed in time $O(Mm^2 + l^2mM) = O(m^3/l + lm^2)$. Setting l to $\sqrt{m}$, this time becomes $O(m^{2.5})$.

Then, C can be computed by

$$C_{ij} = \max_{1 \le k \le N} \{A_{ik} \times B_{kj}\} (i, j = 1, \ldots, N) \tag{32}$$

where the product of submatrices is defined in a similar manner to Equation 31, and the "max" operation is defined on matrices by taking the "max" operation component-wise. Since comparisons and additions of distances are performed in a pair, we can omit counting the number of additions for measurement of complexity. We have to find n^2 maxima of N numbers($O(n^2N)$ time), we have N^3 multiplications of distance matrices in Equation 32. Let us assume that each multiplication of (m, m) matrices can be done in $T(m)$ computing time, assuming that a pre-computed table is available at no cost. The time for constructing the table is given by Lemma 19, and is reasonable when m is small. Then the total computing time is given by

$$O\left(n^2N + N^3T(m)\right) = O\left(n^3/m + (n/m)^3T(m)\right) \tag{33}$$

By Lemma 17 we have, $T(m) = O(m^{2.5})$. Therefore, the time given in Equation 33 becomes $O(n^3/\sqrt{m})$.

5.3.1 Distance matrix multiplication by table-lookup

We divide (m, m)-matrices into strips for $M = m/l$ where $\sqrt{m} \le l \le m$ as follows:

$$A = \left[\boxed{A_1} \ \cdots \ \boxed{A_M} \right] \quad B = \begin{bmatrix} \boxed{B_1} \\ \vdots \\ \boxed{B_M} \end{bmatrix}$$

where A_i is a (m, l) matrix, and B_j is a (l, m) matrix. We regard later A and B as (m, m)-submatrices in Equation 32. Now the product $C = A \times B$ is given by

$$C = \max_{1 \le k \le M} \{A_k \times B_k\} \tag{34}$$

By Lemma 18 we can compute $A_k \times B_k$ in $O(l^2m)$ time, assuming that a pre-computed table is available. Then, the right-hand side of Equation 34 can be computed in time $O(Mm^2 + l^2mM) = O(m^3/l + lm^2)$. Setting l to $\sqrt{m}$, this time becomes $O(m^{2.5})$.

Lemma 17 *The time taken for multiplying (m,m)-submatrices is $O(m^{2.5})$*

Hereafter, we assume $l = \sqrt{m}$. Now we show that for (m,l) matrix A, and (l,m) matrix B, $A \times B$ can be computed in $O(l^2m)$ time.

We assume that the list of numbers $\{(a_{1r} - a_{1s}), \ldots, (a_{mr} - a_{ms})\}(1 \leq r < s \leq l)$ and the list numbers $\{(b_{s1} - a_{r1}), \ldots, (b_{sm} - a_{rm})\}(1 \leq r < s \leq l)$ are already sorted, for all r and s such that $1 \leq r < s \leq l$. Let E_{rs} and F_{rs} be the sorted lists, respectively. For each r and s, we merge list E_{rs} and F_{rs} to form list G_{rs}. This takes $O(l^2m)$ time. The time for sorting is given by Lemma 20. Let H_{rs} be the list of ranks of $(a_{ir} - a_{is}), i = 1, \ldots, m$ in G_{rs}, and let L_{rs} be the list of ranks of $(b_{sj} - b_{rj}), i = 1, \ldots, m$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the i^{th} and j^{th} components of H_{rs} and L_{rs}, respectively. Then we have

$$G_{rs}[H_{rs}[i]] = a_{ir} - a_{is}, G_{rs}[L_{rs}[j]] = b_{sj} - b_{rj}.$$

It will take $O(l^2m)$ time to make lists H_{rs} and L_{rs}, for all r and s. As observed by Fredman [16], we have $a_{ir} + b_{rj} \geq a_{is} + b_{sj}$ or equivalently $a_{ir} - a_{is} \geq b_{sj} - b_{rj}$. He observed that the information of ordering for all i, j, r and s in the right-hand side of the above formula is sufficient to determine the product $A \times B$ by pre-computed decision tree. We observe that to compute all components of $A \times B$ it is enough to know the above ordering for all r and s,

$$a_{ir} - a_{is} \geq b_{sj} - b_{rj} \Leftrightarrow H_{rs}[i] \geq L_{rs}[j] \tag{35}$$

We use a short cut notation $a_{12} \ldots a_{l-1l}$ to express the sequence $a_{12} \ldots a_{1l} a_{23} \ldots a_{2l} \ldots a_{l-1l}$. The list $H[i] = H_{12}[i] \ldots H_{l-1l}[i]$ is encoded into a single integer in lexicographic order [39], and $h(H[i])$ is assumed to represent that integer. Similarly, $h(L[j])$ represents the positive integer for the list $L[i] = L_{12}[i] \ldots L_{l-1l}[i]$. The time for this encoding for all $H[i]$ and $L[j]$ is $O(l^2m)$. Then the pre-computed table I gives the desired index for the inner product, that is,

$$I[h(H[i]), h(L[j])] \;=\; k \Leftrightarrow k \text{ is the index for } \max_{1 \leq k \leq l} \{a_{ik} + b_{ik}\} \tag{36}$$

After the encoding we can compute m^2 inner products in $O(m^2)$ time, assuming that the pre-computed table I is available. Hence $A_k \times B_k$ in Equation 34 is $O(l^2m)$ time.

Lemma 18 *The time taken for computing $A_k \times B_k$ in Equation 34 is $O(l^2m)$*

5.3.2 Computation of table I

Let $H[i] = H_{12}[i] \dots H_{l-1l}[i]$ and $L[i] = L_{12}[i] \dots L_{l-1l}[i]$ be any sequences of $l(l-1)/2$ integers, whose values are between zero and $2m-1$. Let i and j be positive integers representing H and L in lexicographic order, that is, $h(H) = i$ and $h(L) = j$. Then I is defined by

$$I[i,j] = \begin{cases} k, & \text{if there is } k \text{ such that } H_{ks} > L_{ks} \text{ for all } s > k \text{ and } H_{rk} < L_{rk} \text{ for all } r < k, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

This table can be used for table I in Equation 36. The time for computing table I is given by $O\left(l^2(2m)^{2(lC_2)}\right) = O\left(c^{l^2 \log m}\right) = O\left(c^{m \log m}\right)$. for a constant $c > 1$

Lemma 19 *The time taken for construction of table is $O(c^{m \log m})$. where $c > 1$ is a constant.*

5.3.3 Determination of size of submatrices

Let $m = \log n/(\log c \log\log n)$. Then the time for making the table I of this size is easily shown to be $O(n)$, which can be absorbed in the main computing time. Substituting this value of m for $O(m^3/\sqrt{m})$, we have the overall computing time for distance matrix multiplication

$$O\left(n^3(\log\log n/\log n)^{1/2}\right).$$

The time for sorting to obtain the lists E_{rs} and F_{rs} in Section 5.3.1 is $O(m^{2.5}\log m)$. This task of sorting, which is called pre-sort is done for all A_{ij} and B_{ij} in advance, taking

$$O((n/m)^2 m^{2.5}\log m) = O\left(n^2(\log n \log\log n)^{1/2}\right)$$

time where $m = O(\log n/\log\log n)^{1/2}$, which is absorbed into the above main complexity.

Lemma 20 *The total time taken for sorting in Section 5.3.1 is $O(n^2(\log\log n)^{1/2})$.*

5.3.4 Parallelization

We separate the computation in Equation 32 into two parts:

1. Computation of n^2 maxima of N numbers.

2. Computation of N^3 products $A_{ik}B_{kj}(i,j,k=1,\ldots,N)$.

We can find n^2 maxima of N numbers in $O(\log n)$ time with $\frac{n^2N}{\log n} = \frac{n^3}{m\log m} = \frac{n^3 * \log\log n}{\log^2 n}$ processors on a EREW PRAM [39].

In the computation of A_kB_k in Equation 34, there are $O(l^2)$ independent merging of lists of length m. Such computation is done for $k=1,\ldots,m/l$. There are also $M-1$ "max" operations on (m,m)-matrices. It is obvious that there is a parallel algorithm for these operations with $O\left(\frac{m^3}{l}\right) = O(m^{2.5})$ cost, with $O\left(\frac{m*l}{\log m}\right) = O\left(\frac{m^{1.5}}{\log m}\right)$ processors, in $O(m\log m)$ time, where $l=\sqrt{m}$. The tasks of encoding [39] and table-lookup can be done within the above processors and time complexities. Since N^3 products can be computed in parallel, we have a parallel algorithm $A_{ik}B_{kj}$ $P = O\left((n/m)^3 m^{1.5}/\log m\right) = O\left(n^3(\log\log n)^{1/2}/(\log n)^{3/2}\right)$ processors, and $T = O(m\log m) = O(\log n)$ as $m = \Theta(\log n/\log\log n)$.

The lists E_{rs} and F_{rs} have to be broadcast to the computation of N^3 products of $A_{ik}B_{kj}$. This is done in $O(\log n)$ time with

$$O\left(n^3(\log\log n)^{1/2}/(\log n)^{3/2}\right)$$

processors since a datum can be broadcast to N location with $O(N/\log N)$ processors and $O(\log N)$ time. The task of table construct and pre-sort described above in the section can be be done within the above complexities. The complexities of computing products $A_{ik}B_{kj}$ dominates those of maximum computation, and those pre-sort and table construction are much lower.

5.4 All pairs longest paths

In this section, we give a serial algorithm for APLP for directed acyclic graphs with unit edge costs. This algorithm is analogous to the one given by Alon *et al.* [3]. Let $D^{(l)}$ be the l-th approximate matrix for D^* defined by $d_{ij}^{(l)} = d_{ij}^*$ if $d_{ij}^{(l)} \leq l$, and $d_{ij}^{(l)} = -\infty$ otherwise. Then we can compute $D^{(r)}$ by the following algorithm algorithm. Let A be the adjacency matrix.

Algorithm 1: Longest distances by Boolean matrix multiplication

1. $A := \{a_{ij}\}$ where $a_{ij} = 1$ if $d_{ij} = 1$, and 0 otherwise;

2. $B := I$. {Boolean identity matrix }
3. for $l := 2$ to r do begin
4. $B := A \cdot B$. {Boolean matrix multiplication }
5. for $i := 0$ to $n-1$ do for $j := 0$ to $n-1$ do
6. if $b_{ij} = 1$ then $d_{ij}^{(l)} := l$ else $d_{ij}^{(l)} := -\infty$;
7. if $d_{ij}^{(l-1)} = -\infty$ then $d_{ij}^{(l)} := d_{ij}^{(l-1)}$
8. end.

Algorithm 2: Scaling APLD

{Accelerating phase}

1. for $l := 2$ to r do compute $D^{(l)}$ using Algorithm 1;

{Cruising phase}

2. $l := r$;
3. for $s := 1$ to $\lceil \log_{3/2} n/r \rceil$ do begin
4. for $i := 0$ to $n-1$ do
5. Scan the i^{th} row of $D^{(l)}$ and find the smallest set of equal $d_{ij}^{(l)}$'s such that $\lceil l/2 \rceil \le d_{ij}^{(l)} \le l$ and let the set of corresponding indices j be S_i;
6. $l_1 := \lceil 3l/2 \rceil$.
7. for $i := 0$ to $n-1$ do for $j := 0$ to $n-1$ do begin
8. if $S_i \neq \phi$ then $m_{ij} := \max_{k \in S_i} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$ else $m_{ij} = -\infty$
9. if($m_{ij} > l_1$) then $m_{ij} = -\infty$
10. if($m_{ij} = -\infty$) then $d_{ij}^{(l_1)} := d_{ij}^{(l)}$
11. else $d_{ij}^{(l_1)} := m_{ij}$
12. end;
13. $l := l_1$
14. end

Algorithm 2 computes $D^{(l)}$, from $l = 2$ to r in the accelerating phase spending $O(rn^3)$ time, and computes $D^{(l)}$ for $l = r, \lceil \frac{3}{2} r \rceil, \left\lceil \frac{3}{2} \lceil \frac{3}{2} r \rceil \right\rceil, \ldots, n'$ by repeated squaring in the cruising phase, where n' is the smallest integer in this series of l such that $l \ge n$. The key observation in the cruising phase is that we only need to check S_i at line 8, whose size is not larger that $2n/l$. Hence the computing time of one iteration beginning at line 3 is $O(n^3/l)$. Therefore, the time of the cruising phase is given with $N = \lceil \log_{3/2} n/r \rceil$

by

$$O\left(\sum_{s=1}^{N} n^3/((3/2)^s r)\right) = O(n^3/r).$$

Balancing the two phases with $rn^\omega = n^3/r$ the time taken by Algorithm becomes $O\left(n^{(\omega+3)/2}\right)$ with $r = O\left(n^{(3-\omega)/2}\right)$.

When we have a directed graph G whose edges costs are between zero and M where M is a positive integer, we can convert the graph G to $G' = (V, E)$ by adding M auxiliary vertices $v_1, \ldots, v_{M-1}$ for $v \in V$. The edge set is also modified to E'. If $c(v, w) = l$, w is connected from v_{l-1} in E' where $v_0 = v$. Obviously we can solve the problem for G by applying Algorithm 2 to G', which takes $O\left((Mn)^{(\omega+3)/2}\right)$.

5.4.1 Parallelization for graphs with unit costs

We design a parallel algorithm on a EREW PRAM for a directed acyclic graph with unit edge costs. Let A be the adjacency matrix used in Algorithm 1. There is a path from i to j of length less than or equal to l if and only if $A^l(i, j) = 1$, where A^l is the l-th power of A by Boolean matrix multiplication. By repeated squaring, we can get $A^l(l = 1, 2, 4, \ldots, n')$ with $\lceil \log n \rceil$ Boolean matrix multiplications, where n' is the smallest in this series of l such that $l \geq n$. These matrices give a kind of approximate estimation on the path lengths.

Algorithm 3: Longest distances up to 2^R

{Approximation phase}

1. $A^{(1)} := A$; $l := 1$;
2. for $s := 1$ to 2^R do begin
3. $\quad A^{(2l)} := A^{(l)} \cdot A^{(l)}$; {Boolean matrix multiplication}
4. $\quad l := 2l$
5. end;

{Gap filling phase}

6. $D^{(1)} := D$.
7. for $s := 1$ to R do begin
8. $\quad$ for $l := 2^{s-1} + 1$ to 2^s do begin
9. $\quad\quad A^{(l)} := A^{(l-2^{s-1})} \cdot A^{(2^{s-1})}$; {Boolean matrix multiplication}
10. $\quad\quad$ for $i := 1$ to $n - 1$ do for $j := 0$ to $n - 1$ do

11. if $a_{ij}^{(l)} = 1$ then $b_{ij}^{(l)} := l$ else $b_{ij}^{(l)} := -\infty$
12. end;
13. for $i := 1$ to $n-1$ do for $j := 0$ to $n-1$ do
14. $d_{ij}^{(l)} := \max_{2^{s-1}<l\leq 2^s}\{b_{ij}^{(l)}\}$
15. end.

The computing time of this algorithm is $O(2^R n^\omega)$. If we let $R = \lceil \log_2 n \rceil$, we can solve the APLD but then time becomes $O(n^{\omega+1})$, which is not efficient. We substitute Algorithm 3 for Algorithm 1 in Algorithm 2, and call the resulting algorithm Algorithm 2' and let $2^R = n^{(\omega-\omega)/2}$, however, we can solve the APLD problem in $O(n^{(\omega+3)/2})$ time. Which is on par with Algorithm 2. We can perform 2^{s-1} multiplications in parallel at line 9. Also we can compute 2^{s-1} matrices $B^{(l)}$ at lines 10 and 11 in parallel. At line 14, we can find the maximum in $O(s)$ time with $O(2^s/s)$ processors.

In cruising phase, we can find the maximum at line 8 in $O(\log n)$ time, with $O(n/(l\log n))$ processors. The rest is absorbed in these complexities. The summary of complexities is as follows

Accelerating phase $T = O(R\log n), P = (n^w.2^R)$

Cruising phase $T = O(\log(n/2^R).\log n), P = O(n^3/(2^R \log n))$

If we let $2^R = n^{(3+\omega)/2}/\sqrt{\log n}$, we have the overall complexity as follows:

$$T = O(\log^2 n), \quad P = n^{(3+w)/2}/\sqrt{\log n}$$

If we have a graph with edges costs up to M we can replace n by Mn in the above complexities.

5.4.2 Parallelization for graphs with general costs

If edges costs are non-negative real numbers, we can apply the techniques in the previous sections. We use the longest distance matrix multiplication algorithm described in Section 5.4, and slightly modify the cruising phase [40]. In Algorithm 2, there is no difference between distances and lengths of paths since the edge costs are ones. In line 5 of Algorithm 2, we choose set S_i based on the distances $d_{ij}^{(l)} (j = 0, \ldots, n-1)$ satisfying $\lceil l/2 \rceil \leq d_{ij}^{(l)} \leq l$ to guarantee the correct computation of distances on path lengths, not distances. If we keep track of path lengths, we can adopt Algorithm 2 here. The definition of $d_{ij}^{(l)}$ here is that it gives the cost of the longest path whose length is not

greater than l. The algorithm follows with a new data structure, array $Q^{(l)}$, such that $q_{ij}^{(l)}$ is the length of a path that gives $d_{ij}^{(l)}$.

Algorithm 4

{Accelerating phase}

1. $l := 1; D^{(1)} := D; Q^{(1)} := \{q_{ij}^{(1)}\}$, where $q_{ij}^{(1)} = \begin{cases} 0, & \text{if} i = j \\ 1, & \text{if} i \neq j \text{and} (i,j) \in E \\ -\infty, & \text{otherwise;} \end{cases}$
2. for $s := 1$ to $\lceil \log_2 r \rceil$ do begin
3. $D^{(2l)} := D^{(l)} \times D^{(l)}$; {distance matrix multiplication}
4. $Q^{(2l)} := \{q_{ij}^{(2l)}\} = \begin{cases} q_{ik}^{(l)} + q_{kj}^{(l)}, & \text{if } d_{ij}^{(2l)} \text{ is updated by } d_{ik}^{(l)} + d_{kj}^{(l)} \\ q_{ij}^{(l)}, & \text{otherwise;} \end{cases}$
5. $l := 2l$
6. end;

{Cruising phase}

7. for $s := 1$ to $\lceil \log_{3/2} n/r \rceil$ do begin
8. for $i := 0$ to $n - 1$ do
9. Scan the i^{th} row of $Q^{(l)}$ and find the smallest set of equal $q_{ij}^{(l)}$'s such that $\lceil l/2 \rceil \leq q_{ij}^{(l)} \leq l$ and let the set of corresponding indices j be S_i;
10. $l_1 := \lceil 3l/2 \rceil$;
11. for $i := 0$ to $n - 1$ do for $j := 0$ to $n - 1$ do begin
12. if $S_i \neq \phi$ then begin
13. $m_{ij} := \max_{k \in S_i} \{d_{ik}^{(l)} + d_{kj}^{(l)}\}$;
14. $k :=$ one that gives the above maximum and satisfies that $q_{ik}^{(l)} + q_{kj}^{(l)}$ is maximum among such k;
15. $L := q_{ik}^{(l)} + q_{kj}^{(l)}$;
16. end else $\{S_i = \phi\} L := -\infty$; $m_{ij} = -\infty$
17. $d_{ij}^{(l_1)} = m_{ij}; q_{ij}^{(l)} = L$
18. if $d_{ij}^{(l)} = -\infty$ then begin $d_{ij}^{(l_1)} := d_{ij}^{(l)}; q_{ij}^{(l_1)} := L$ end
19. $l := l_1$
20. end
21. end.

We index time T and the number of processors P in the accelerating phase and

cruising phase by 1 and 2. Then we have

$$T_1 = O(\log r \log n), \quad P_1 = O(n^3(\log\log n)^{1/2}/(\log n)^{3/2}.$$

The computation of all S_i can be done in $O(\log n)$ time with $O(n^2)$ processors. The dominant complexity in the cruising phase is at line 13. This part can be computed in $O(\log(n/r))$ time with $O((n/r)/\log(n/r))$ processors. Thus we have

$$T_2 = O(\log n \log(n/r)), \quad P_2 = O(n^2(n/r)/\log(n/r)).$$

Letting $r = (\log n/\log\log n)^{3/2}$ yields

$$T_1 = O(\log n \log\log n), \quad P_1 = O(n^3(\log\log n)^{1/2}/(\log n)^{3/2}).$$

$$T_2 = O(\log^2 n), \quad P_2 = O(n^3(\log\log n)^{3/2}/(\log n)^{5/2}).$$

Thus the cost is given by

$$\begin{aligned} P_1T_1 + P_2T_2 &= O(n^3(\log\log n)^{3/2}/(\log n)^{1/2}) + O(n^3(\log\log n)^{3/2}/(\log n)^{1/2}) \\ &= O(n^3(\log\log n)^{3/2}/(\log n)^{1/2}) \\ &= o(n^3) \end{aligned}$$

The APLP problem can also solved with in the above stated complexities. The only thing we need is to keep track of witnesses at distance matrix multiplication, and the maximum operation at line 13. Since the operation broadcasting N items can be done in $O(1)$ time on a minimum-CRCW PRAM using $O(N)$ processors, and operation of finding minimum among N numbers can be done in $O(1)$ with $O(N)$ processsors on a minimum-CRCW PRAM or $O(\log\log n)$ time with $O\left(\frac{N}{\log\log N}\right)$ on a common-CRCW PRAM. The time will be $O(\log n)$ on a minimum-CRCW PRAM and $O(\log n \log\log n)$ time on common-CRCW PRAM. The above algorithm will $o(n^3)$ cost on a minimum-CRCW PRAM.

Chapter 6

Scheduling Interval Ordered Tasks In Parallel

6.1 Introduction

The problem of scheduling unit execution time tasks on a number of processors under arbitrary constraints has been studied extensively in past. When m, the number of processors is part of input, the problem is $\mathcal{NP}$-hard[42]. There are a number of efficient algorithms for the case when $m = 2$ [11]. But, the problem problem is still open for the 3-processor case [5]. Moreover, polynomial time algorithms are there, when m is part of the input, and the precedence constraints are trees [25] or interval orders [32]. The main algorithmic tool employed in obtaining polynomial time sequential algorithms for solving these problem is known as *list scheduling*. Briefly the method is as follows:

> Form a priority list of tasks and construct a schedule iteratively by choosing a maximal set of $r \leq m$ independent tasks (tasks with no precedence constraints with in them) of highest priority in each iteration.

Sunder *et al.* have shown that *list scheduling problem* is $\mathcal{P}$-complete, and hence it is unlikely to be parallelizable. Helmbold and Mayr [22] showed that the construction of the list schedule (with $m = 2$, arbitrary execution times for tasks and empty precedence constraints) is $\mathcal{P}$-complete, and hence unlikely to be parallelizable. However, parallel algorithms are known for the 2-processor case. Helmbold and Mayr [23] presented the first $\mathcal{NC}$ algorithm, and Vazirani and Vazirani [43] presented an $\mathcal{RNC}$ algorithm for the

problem. Sunder *et al.* [38] have proposed the first $\mathcal{NC}$ algorithm for scheduling the *interval ordered tasks* on a priority-CRCW PRAM. Their algorithm runs in $O(\log^3 n)$ time with $O(n^3 \log n)$ cost.

In this chapter, we present an $\mathcal{NC}$ algorithm for scheduling interval ordered tasks on m machines. The sequential algorithm for this problem, proposed by Papadimitriou *et al.* is based on list scheduling method [32]. Our algorithm makes use of structural properties of interval orders, and some techniques developed by Bartusch *et al.* [5]. Our parallel algorithm for scheduling interval orders constructs the same schedule as produced by the sequential list scheduling algorithm.

6.2 Basic Definitions

Let $G = (V, A)$ be a partial order (or equivalently a transitive acyclic directed graph) consisting of $n = |V|$ nodes. We refer to G as a *precedence constraint graph* and the elements of V as tasks. A node u is a *successor* of a node v, if there is a directed path from v to u in G. The set of successors of v is denoted by $N_G(v)$ (or simply $N(v)$ if the context is clear), v is a *maximal node* if $N(v) = \phi$.

A *schedule* of length t for G on m processors is an $m \times t$ matrix S, where the columns are indexed from $1, \ldots, t$ and the rows are indexed from $1, \ldots, m$. Each task x is assigned to an unique entry $(p(x), t(x))$ in S such that $(x, y) \in A$ implies $t(x) < t(y)$. For any task $x \in V$, the entry $(p(x), t(x))$ denotes that task x is scheduled on processor $p(x)$ at time instant $t(x)$. No two tasks are assigned to same entry in S. The length of S is denoted by $|S|$. An entry in S is also called a *slot*. A slot of S to which no task is assigned is said to be *empty*. Two schedules S_1 and S_2 for G are considered to be the same if for every task x in G, the column assigned to x in S_1 is same as the column assigned to x in S_2. This is because the processors are identical; it is irrelevant which processor is actually assigned to the task. We denote the *subschedule* of S consisting of columns $i, i+1, \ldots, j (1 \leq i \leq j \leq t)$ by $S[i, j]$. Let $S' and S''$ be two schedules of size $m \times t_1$ and $m \times t_2$ for two partial orders G_1 and G_2. The *concatenation* of S' and S'', denoted by $S' \circ S''$, is the schedule of size $m \times (t_1 + t_2)$ obtained by concatenating the two matrices S' and S''. Given a precedence constraint graph G, a list L in G is said to be *precedence preserving* if for any vertices u and v in G, u is successor of v implies

that v precedes u in L.

Definition 21 [32] *An* interval order *is partial order $G = (V, A)$ where V is a set of intervals on the real line and $(u, v) \in A$ iff $x \in u, y \in v$ implies $x < y$.*

The left and the right endpoint of an interval $u \in V$ is denoted by $l(u)$ and $r(u)$, respectively. We say u precedes v if $(u, v) \in A$. We assume that, without loss of generality, that all endpoints are distinct. For each $v \in V$, clearly neighbors of v in G, $N_G(v) = \{u \in V \mid l(u) > r(v)\}$. The interval order corresponding to a list L of intervals is denoted by $G(L)$. For any two list L_1 and L_2, let $L_1 \cdot L_2$ denote the *concatenation* of the two lists The cardinality of a list L is denoted by $|L|$. For $1 \leq i \leq j \leq |L|$, $L(i)$ denotes the i^{th} element of L and $L(i, \ldots, j)$ denotes the sublist of L consisting of the elements $L(i), L(i+1), \ldots, L(j)$.

Let L be a list of intervals and let $u \notin L$ be an arbitrary interval. Let S be the schedule constructed from $G(L)$ with m processors. A column in S is *incomplete* if less than m tasks are scheduled in it. We say an incomplete column c in S is *feasible* for u if the column c does not contain any interval that precedes u in any column $c' > c$. An empty slot in an incomplete column which is feasible for u is said to be *available* for u.

6.3 Sequential Algorithm

In this section, we first describe the *List scheduling* algorithm proposed by Coffman [10].

Algorithm: List-Schedule(G, L, m).

Inputs: An arbitrary precedence constraint graph $G = (V, A)$, a precedence-preserving list L of the tasks V, and m the number of processors.

1. $t = 1$;
2. while $L \neq \emptyset$ do
   (a) Initialize an empty set L'.
   (b) Put the first node v of L into L'. Scan L form left to right. When a node w is scanned, if w is independent with every node in L' (namely, for any $u \in L'$, $(v, w) \notin A$, and $(w, u) \notin A$), then include w into L'. Repeat this process until either L' contain m nodes or all nodes of L are considered.
   (c) Schedule the nodes in L' in column t.

(d) $L = L - L', t = t + 1$.

3. Output the schedule constructed.

The *list scheduling problem* for an arbitrary precedence graph $G = (V, A)$, precedence-preserving list L and the number of processors m, is to compute List-Schedule(G, L, m). Sunder *et al.* [38] have shown this is $\mathcal{P}$-complete under NC reduction. We now describe the sequential algorithm for the scheduling interval ordered tasks, proposed by Papadimitriou *et al.* [3]. They have described an implementation that runs in $O(|V| + |A|)$ time.

Algorithm: Sequential-Schedule(L, m).

Inputs: L, a list of intervals representing the interval order $G = (V, A)$, and the number of processors m.

1. Sort the list L of intervals by increasing order of right end points.
2. Compute S = List-Schedule$(G(L), L, m)$.

6.4 Parallel Algorithm

In this section, we present an $\mathcal{NC}$ algorithm for scheduling interval orders. Our algorithm is based on the $\mathcal{NC}$ algorithm proposed by Sunder *et al.* [38]. They have reduced the problem of constructing the optimal schedule to the problem of computing the optimal schedule length and gave an algorithm for computing the optimal schedule length, that runs in $O(\log n)$ time with $O(n^3)$ processors on a priority-CRCW PRAM. In this section, we present an $O(\log n)$ time algorithm with $o(n^3)$ cost on a minimum-CRCW PRAM for the same problem and use their parallel algorithm for constructing optimal schedule.

6.4.1 Computing optimal schedule length

Bartusch *et al.* [5] defined a *bounding function* b on the node set V, $b(v)$ is an upper bound on the column number by which the node v must be scheduled in any schedule of length t for G. They have also shown that if there is a schedule of length at most t for G on m processors it can be constructed by forming a list L of the nodes sorted by nondecreasing order of the bounds and then computing the list schedule associated with L. The representation theorem of Bartusch *et al.* [5]:

Theorem 7 [5] *There exists a schedule for the interval order G on m processors with length t iff for every i with $1 \leq i \leq t$, $|\{u \mid b(u) \leq i\}| \leq m * i$.* ∎

Lemma 21 [38] *Let x and y be two nodes in an interval order G.*

(1) If $r(x) < r(y)$ then $b(x) \leq b(y)$.

(2) If $b(x) < b(y)$ then $r(x) < r(y)$. ∎

The bounding function b cannot be computed in parallel. Sunder *et al.* later defined another bounding function b' such that Theorem 7 holds with respect to b' for interval ordered tasks and can be computed in parallel. The algorithm described by them is as follows [38]:

Algorithm: Bound$(G = (V_G, E_G), m, t)$.

1. Define a directed acyclic graph $D = (V_D, E_D, w)$ with integral edge weights w as follows:
   (a) $V_D = V_G \cup \{s\}$ where s is a new sink node.
   (b) $E_D = E_G \cup \{(v, s) \mid v \in V_G\}$.
   (c) Define $w(v, s) = 0$ for all $v \in V_D$.
   (d) For every $(v, u) \in E_D$ such that $u \neq s$ do compute
   $$S(v, u) = \{x \in N_G(v) \mid r(x) \leq r(u)\},$$
   $$w(v, u) = \lceil |S(v, u)|/m \rceil$$
2. for all $v \in V_D$, find $d(v)$, the length of the longest path in D from v to sink s.
3. for every $v \in V_G$, compute $b'(v) = t - d(v)$.

The complexity of the above algorithm is dominated by Step 2 that involves the computation of all pairs longest distance, of a directed acyclic graph. For this step Sunder*et al.* have proposed an algorithm that takes $O(\log n)$ time with $O(n^3 \log n)$ cost. But, this step can be done in $O(\log n)$ time using $O(n^3(\log\log n)^{3/2}/(\log n)^{1/2}) = o(n^3)$ cost using algorithm presented in Chapter 5, on a minimum-CRCW PRAM.

Lemma 22 [38] *We can replace $b(v)$ by $b'(v)$ in Theorem 7* ∎

The following algorithm [38] computes the length of a optimal schedule for an interval order G.

Algorithm: Length(L, m).

1. As in the algorithm Bound, construct the edge weighted graph $D = (V_D, A_D, w)$ from $G(L)$, and compute, for each node u of $G(L)$, the longest distance $d(u)$ in D.
2. In parallel, for every $1 \leq t \leq n$, check if the following conditions are true: for every $1 \leq i \leq t, |\{u \mid b'(u) = t - d(u) \leq i\}| \leq m * i$.
3. Output the smallest t such that the foregoing conditions are true.

The complexity of the algorithm is dominated by Step 1 and it takes $O(\log n)$ time with $o(n^3)$ cost. The Steps 2 and 3 take at most $O(\log n)$ time with $O(n^2)$ processors.

6.4.2 Parallel algorithm for constructing optimal schedule

In this section, we describe a parallel algorithm [38] that constructs the list schedule for an interval order $G = (V, A)$. Let L be the list of the tasks of V in increasing order of right end points, and $SeqS$ = Sequential-Schedule (L, m) = List-Schedule$(G(L), L, m)$. Their parallel algorithm constructs $SeqS$ using divide and conquer technique. Suppose $\|SeqS\| = t$ and $t_1 = \lfloor t/2 \rfloor$. Let $S' = SeqS[1, t_1]$ and $S'' = SeqS[t_1 + 1, t]$, construct S' and S'' in parallel. By Lemma 21, there exists an integer $i \leq n$ such that all tasks in $L(1, \ldots, i)$(and possible some other tasks) are scheduled in the columns of S' and each each column of S' contains at least one task from $L(1, \ldots, i)$ [38]. Sunder and He have described algorithm to find all tasks those can be scheduled in t_1. The algorithm proposed in [38] for finding optimal schedule is as follows:

Algorithm: Parallel-Schedule(L, m)**.**

1. For $i := 1$ to n do
   Compute Length$(L(1, \ldots, i), m)$.
   Let t=Length$(L(1, \ldots, n), m)$
2. if $t = 1$, then schedule all tasks of L in one column and return else, let s be the integer such that Length$(L(1, \ldots, s), m) \leq t/2$ and Length$(L(1, \ldots, s+1), m) > t/2$
3. $L_1 = L(1, \ldots, s)$, $L_2 = L(s + 1, \ldots, n)$
4. Let u be the last interval in L_1. Define $L_3 = \{x \in L_2 \mid l(x) < r(u)\}$. $\{L_3$ is the set of intervals in L_2 that might be scheduled with the intervals in L_1. These are pairwise incomparable$\}$

5. X = Jump(L_1, L_3, m) { Jump is a function defined later. It returns the subset X of L_3 that might be scheduled with L_1. }
6. $L' = L_1 \cdot X$ and $L'' = L_2 - X$.
7. S' = Parallel-schedule(L', m) and S'' = Parallel-schedule(L'', m).
8. Output $S' \circ S''$

Let $L_1 = L(1, \ldots, s), L_2(s+1, \ldots, n)$, and let $u = L(s)$. Let $SeqS_1$ = List-Schedule(L_1, m). Let $\|SeqS_1\| = t_1$. Then, let us define the following sets for some $x \in L_2$ that occurs at position k of L.

$A(x) = \{y \in L(s+1, \ldots, k-1) \mid l(y) < l(x) < r(u) < r(y) < r(x)\}$.

$B(x) = \{y \in L(s+1, \ldots, k-1) \mid l(x) < l(y) < r(u) < r(y) < r(x)\}$.

$C(x) = \{y \in L(s+1, \ldots, k-1) \mid r(u) < l(y) \text{and} r(y) < r(x)\}$.

$A(x), B(x)$, and $C(x)$ are sorted by increasing order of right end points.

Algorithm: Jump(L_1, L_3, m)

1. for every $x \in L_3$ do
   compute $A(x)$ and $B(x)$.
2. for every $x \in L_3$ do
   (a) $S_1(x)$ = the number of empty slots available for x in $SeqS_1[c(x), t_1]$,
   (b) $S_2(x)$ = the number of empty slots available for x in $SeqS_1(x)[c(x), t_1]$, { $SeqS_1(x)$ = List-Schedule$(L_1.A(x), m)$ }
3. for every $x \in L_3$,
   let $B(x) = \{b_1^x, \ldots, b_{k_x}^x\}$ sorted by increasing order of $left$ end points.
4. return $X = \{x \mid (\exists j : 1 \le j \le k_x : S_2(x) - S_1(b_j^x) \ge j) \text{or} (S_2(x) > k_x)\}$.

We now describe, how Step 2 of Jump can be implemented [38]. For $1 \le i \le n$, define the list $C(1, i) = \{c_1, \ldots, c_i\}$ where $c_j (1 \le j \le i)$ is a copy of the interval x. Because $S_1(x)$ is the number of empty slots in $SeqS_1[c(x), t_1]$, evaluating $S_1(x)$ is equivalent to finding the largest $i \le n$ such that $C(x, i)$ can be scheduled in empty slots of $SeqS_1[c(x), t_1]$. Hence Length$(L_1, C(x, i), m) > t_1$ iff $S_1(x) < i$. Similarly because $S_2(x)$ is the number of empty slots in $SeqS_1(x)[c(x), t_1]$, evaluating $S_2(x)$ is equivalent to finding finding the largest $i \le n$ such that the intervals in $C(x, i)$ can be scheduled in empty slots in $SeqS_1(x)[c(x), t_1]$. There are two cases.

Case 1. Suppose $\|SeqS_1(x)\| > t_1$. Then, at least one node in $A(x)$ is scheduled in

$SeqS_1(x)$ in a column $c' > t_1$. Because every node in $A(x)$ must be scheduled before the node x, there are no empty slots available for x in $SeqS_1(x)[c(x), t_1]$. Hence $S_2(x) = 0$.

Case 2. Suppose $\|SeqS_1(x)\| = t_1$. We can evaluate $S_2(x)$ using the following condition: Length$(L_1.A(x).C(x,i), m) > t_1$ iff $S_2(x) < i$.

Thus for each $x \in L_3$, $S_1(x)$ and $S_2(x)$ can be computed by either making $O(n)$ parallel calls to procedure Length or $O(\log n)$ calls in sequence by doing a binary search [38]. Sunder *et al.* have stated that above algorithm takes $O(\log^3 n)$ time with $O(n^4 \log n)$ or alternatively $O(\log^2 n)$ time with $O(n^5 \log n)$ cost. We first perform a better analysis of the algorithm and reduce the cost by a factor of $O(\log n)$ in both the implementations. Then, we use a better algorithm for computing the length of the schedule as described in Section 6.4.1, and further reduce the cost by another factor of $\log n$.

Analysis of Algorithm.

- Setting L_1 and L_2 in Step 3 requires to n parallel calls to the algorithm Length (in Step 1) [38]. If C and T are the cost and time required for the algorithm Length. Then, the cost of Steps 1 to 3 is $O(nC)$ and time required is $O(T)$.
- For setting up L' and L'' we are calculating X in Step 5 i.e. calling algorithm Jump. In the algorithm Jump, $S_1(x)$ and $S_2(x)$ are calculated for each $x \in L_3$ by making $O(\log n)$ sequential calls (by doing an exponential and binary search) to algorithm Length or $O(n)$ parallel calls [38]. Thus, we can observe that the cost and time required for the algorithm Jump is $(nC \log n)$ and $O(T \log n)$ or $O(n^2 C)$ and $O(T)$.
- If we sum up all the costs and times in the first recursive step then, the time, and cost of the algorithm Parallel-Schedule are $O(T \log n)$ and $O(nC \log n)$ or alternatively $O(T)$ and $O(n^2 C)$.
- We use the all pairs longest path algorithm described in Chapter 5 in the algorithm Length. Then, the algorithm Length takes $T = O(\log n)$ time with $C = O(n^3 (\log \log n)^{3/2}/(\log n)^{1/2}) = o(n^3) = \Omega(n^{2+\epsilon})$ cost on a minimum-CRCW PRAM.
- Thus the algorithm Parallel-Schedule takes $O(\log^2 n)$ time with $o(n^4 \log n)$ cost which is also $\Omega(n^3 \log n)$ or $O(\log n)$ time with $o(n^5)$ cost, which is also $\Omega(n^4 \log n))$, in the first recursive step.

Theorem 8 *The list scheduling problem for interval order can be solved in* $O(\log^3 n)$ *time with* $O(n^4(\log\log n)^{3/2}(\log n)^{1/2})$ *cost or* $O(\log^2 n)$ *time with* $O\left(\frac{n^5(\log\log n)^{3/2}}{(\log n)^{1/2}}\right)$ *cost on a minimum-CRCW PRAM.*

Proof: The correctness of proof follows from [38]. Step 1 ensures the recursion depth of Parallel-Schedule is $O(\log n)$ because each recursive call reduces the schedule length by a factor 2. The sizes of S' and S'' are at most $n/2$. Let $C(n)$ denotes the cost of the parallel algorithm, then we have the following recurrence relations.

$$\begin{aligned}
C(n) &= 2C\left(\tfrac{n}{2}\right) + n\log n f(n), \\
&\quad \text{where } f(n) \text{ is the cost of finding the optimal schedule length} \\
&\quad \text{and } f(n) = O(n^3(\log\log n)^{3/2})/(\log n)^{1/2}) = \Omega(n^2) \\
&= 2^2 C\left(\tfrac{n}{2^2}\right) + 2\tfrac{n}{2}\log\left(\tfrac{n}{2}\right) f\left(\tfrac{n}{2}\right) + n\log n f(n) \\
&\leq 2^2 C\left(\tfrac{n}{2^2}\right) + n\log n f\left(\tfrac{n}{2}\right) + n\log n f(n) \\
&\leq 2^3 C\left(\tfrac{n}{2^3}\right) + 2^2\left[\tfrac{n}{2^2}\log\left(\tfrac{n}{2^2}\right) f\left(\tfrac{n}{2^2}\right)\right] + n\log n f\left(\tfrac{n}{2}\right) + n\log n f(n) \\
&\leq 2^3 C\left(\tfrac{n}{2^3}\right) + n\log n f\left(\tfrac{n}{2^2}\right) + n\log n f\left(\tfrac{n}{2}\right) + n\log n f(n) \\
&\leq 2^k C\left(\tfrac{n}{2^k}\right) + n\log n\left[f(n) + f\left(\tfrac{n}{2}\right) + \ldots + f\left(\tfrac{n}{2^{k-1}}\right)\right]
\end{aligned}$$

Since $f(n) = \Omega(n^2)$, $f\left(\frac{n}{2}\right) \leq \frac{1}{2^2} f(n)$. Thus $\left[f(n) + f\left(\frac{n}{2^2}\right) + \ldots + f\left(\frac{n}{2^{k-1}}\right)\right] = \Theta(f(n))$ (See Master's theorem [14])

Since there are $O(\log n)$ recursive calls, the total time taken by the algorithm is $O(\log^3 n)$ time with $o(n^4 \log n)$ (or $O(n^4(\log\log n)^{3/2}(\log n)^{1/2})$) cost. In a similar fashion, we show below that the algorithm Parallel-Schedule takes $O(\log^2 n)$ time with $o(n^5)$ $\left(\text{or } O\left(\frac{n^5(\log\log n)^{3/2}}{(\log n)^{1/2}}\right)\right)$ cost.

$$\begin{aligned}
C(n) &= 2C\left(\tfrac{n}{2}\right) + n^2 f(n), \\
&\quad \text{where } f(n) \text{ is the cost of finding the optimal schedule length} \\
&\quad \text{and } f(n) = O(n^3(\log\log n)^{3/2})/(\log n)^{1/2}) = \Omega(n^2) \\
&= 2^2 C\left(\tfrac{n}{2^2}\right) + 2\tfrac{n^2}{2^2} f\left(\tfrac{n}{2}\right) + n^2 f(n) \\
&\leq 2^2 C\left(\tfrac{n}{2^2}\right) + \tfrac{n^2}{2} f\left(\tfrac{n}{2}\right) + n^2 f(n) \\
&\leq 2^3 C\left(\tfrac{n}{2^3}\right) + 2^2\left[\tfrac{n^2}{2^4} f\left(\tfrac{n}{2^2}\right)\right] + \tfrac{n^2}{2} f\left(\tfrac{n}{2}\right) + n^2 f(n) \\
&\leq 2^3 C\left(\tfrac{n}{2^3}\right) + \tfrac{n^2}{2^2} f\left(\tfrac{n}{2^2}\right) + \tfrac{n^2}{2} f\left(\tfrac{n}{2}\right) \\
&\leq 2^k C\left(\tfrac{n}{2^k}\right) + f(n)\left[n^2 + \left(\tfrac{n^2}{2}\right) + \ldots + \left(\tfrac{n^2}{2^{k-1}}\right)\right] \\
&\leq n^2 f(n)
\end{aligned}$$

∎

Chapter 7

Conclusions

In this thesis we have proposed improved parallel and sequential algorithms for different optimization problems. For the problem of *all pairs shortest routing* considered, we have reduced the cost of existing parallel algorithm proposed by Gupta and Krishnamurti [21], from $O(n^4 \log^3 n)$ to $O(n^4 \log n)$. We have also improved the $O(n^2)$ time S-T routing sequential algorithm proposed by them to $O(m + n \log n)$ time. For the single vehicle routing problem with deadline time constraint on locations that are on a line, and for the single vehicle routing problem with release time constraint on locations that are on a line, we have improved the cost of existing algorithms given by Gupta and Krishnamurti [21], from $O(n^8 \log^3 n)$ to $O(n^4 \log n)$. For the TRP-line problem, we have proposed a new sequential algorithm that takes $O(n^2)$ time and a parallel algorithm that takes $O(\log^2 n)$ time, with $O(n^3 \log n)$ cost. We have also proposed $O(n)$ time optimal parallel algorithms for dSVRPTW-line, rSVRPTW-line and TRP-line problems.

To the best of our knowledge, we do not know any parallel algorithm for the coalescing operations with precedence constraints in real-time systems. However, there are sequential algorithms for this problem [30, 9]. We have proposed an $O(\log^2 n)$ time parallel algorithm, with $O(n^3 \log n)$ cost for this problem. Moreover, we have presented an $O(n)$ time optimal parallel algorithm for this problem.

We have proposed sub-cubic cost algorithms for the *all pairs longest path problem* in directed acyclic graph, by modifying the all pairs shortest path algorithms proposed by Takaoka[40]. For the problem of scheduling interval ordered tasks, we have improved

the cost of existing parallel algorithms proposed by Sunder *et al.* [38] by a $O(\log^2 n)$ factor.

In this thesis we have considered only single vehicle routing problems. It will be interesting to see multiple vehicle routing for VRPTW-line and TRP-line problems and for architectures different from line. For scheduling interval ordered tasks we have described an $\mathcal{NC}$ algorithm. It will also be interesting to see if randomization can be used to get better algorithms.

Bibliography

[1] AFRATI, F., COSMADAKIS, A., PAPADIMITRIOU, C., AND PAPAKONSTANTINOU, N. The complexity of the traveling repairman problem (as cited by TSITSIKLIS[41]). *Theory of Information Applications 20*, 1 (1986), 79–87.

[2] AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. D. *Network Flows.* Prentice-Hall Englewood Cliffs, 1993.

[3] ALON, N., GALIL, Z., AND MARGALIT, O. On the exponent of the all pairs shortest path problem. In *Proceedings of 32nd IEEE FOCS* (1991), pp. 569–575.

[4] BALCÁZAR, J. L., DÍAZ, J., AND GABARRÓ, J. *Structural Complexity*, vol. 1. Springer-Verlag, 1988.

[5] BARTUSCH, M., MOHRING, R. H., AND RADERMACHER, F. J. M-machine unit time scheduling: A report of ongoing research. *Lecture Notes in Economics and Mathematical Systems 304* (1988), 165–212.

[6] BERKMAN, O., SCHIEBER, B., AND VISHKIN, U. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms 14* (1993), 344–370.

[7] BIHARI, T. E., AND GOPINATH, P. Object-oriented real-time systems: Concept and examples. *IEEE computation 25* (1992), 25–32.

[8] BODIN, L., GOLDEN, B., ASSAD, A., AND BALL, M. Routing and scheduling of vehicles and crews: The state of the art. *Computer Operations Research 10* (1983), 62–212.

[9] CHEN, M. I., CHUNG, J. Y., AND LIN, K. J. Scheduling algorithm for coalesced operations in real-time systems. In *Proceedings of the COMPSAC 89* (1989), pp. 143–150.

[10] COFFMAN, E. G., Ed. *Computer and Job Scheduling Theory.* Wiley, New York, 1978.

[11] COFFMAN, E. G., GAREY, M. R., AND JOHNSON, D. S. Approximation algorithms for bin packing: An updated survey. In *Algorithm Design for Computer System Design*, G. Suseillo, M. Lucertini, and P. Serafini, Eds. Springer-Verlag, New York, 1984.

[12] COOK, S. A. A taxonomy of problems with fast parallel algorithms. *Information*

and Control 64 (1985), 2–22.

[13] COPPERSMIT, D., AND WINOGRAD, S. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation 9* (1990), 251–280.

[14] CORMEN, LEISERSON, AND RIVEST. *Algorithms.* MIT Press, 1990.

[15] FORTUNE, S., AND WYLLIE, J. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on the Theory of Computing* (1978), pp. 114–118.

[16] FREDMAN, M. L. New bounds on the complexity of the shortest path problem. *SIAM Journal of Computing 5* (1976), 83–89.

[17] FREDMAN, M. L., AND TRAJAN, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 3 (1987), 596–615.

[18] GAREY, M., AND JOHNSON, D. *Computers and intractability: A guide to theory of $\mathcal{NP}$-completeness.* Addison Wesely, 1991.

[19] GAREY, M. R., AND JOHNSON, D. S. Two-processor scheduling with start times and deadlines. *SIAM Journal of Computing 6* (1977), 416–426.

[20] GOLUMBIC, M. C. *Algorithmic Graph Theory.* Academic Press, 1980.

[21] GUPTA, A., AND KRISHNAMURTI, R. Parallel algorithms for vehicle routing problems. In *Proceedings of High Performance Computing* (1997), IEEE, pp. 144–151.

[22] HELMBOLD, D., AND MAYR, E. Fast scheduling algorithms on parallel computers. In *Advances in Computing research.* JAI Press, London, 1987.

[23] HELMBOLD, D., AND MAYR, E. Two processor scheduling is in $\mathcal{NC}$. *SIAM Journal of computing 16* (1987), 747–759.

[24] HOROWITZ, E., AND SAHANI, S. *Fundamentals of computer algorithms.* Galgotia publications, 1984.

[25] HU, T. C. Parallel sequencing and assembly line problems. *Operations Research 9* (1961), 841–848.

[26] JAJA, J. *An Introduction to Parallel Algorithms.* Addison Wesely, 1991.

[27] KARUNO, Y., NAGAMOCHI, H., AND IBARAKI, T. Vehicle scheduling on a tree with release and handling times. In *Proceedings of the Fourth International*

Symposium on Algorithm and Computation (1993), Springer-Verlag, pp. 486–495.

[28] KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. The power of parallel prefix. *IEEE Transactions on Computers 34* (1985), 307–363.

[29] LENSTRA, J. K., KAN, A. H. G. R., AND BRUCKER, B. Complexity of machine scheduling problems. *Annals of Discrete Mathematics 1* (1977), 343–362.

[30] LIU, L. T., CHEN, G. H., AND LIN, K. J. An algorithm for coalescing operations with precedence constraints in real-time systems. *Information Processing Letters 46* (1993), 129–133.

[31] PAPADIMITRIOU, C. The Euclidean traveling salesman problem is $\mathcal{NP}$-complete. *Theoretical Computer Science 4* (1977), 237–244.

[32] PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. Scheduling interval-ordered tasks. *SIAM Journal of Computing 8*, 3 (1979), 405–409.

[33] PSARAFTIS, H., SOLOMON, M., MAGNANTI, T., AND KIM, T. Routing and scheduling on a shoreline with release times. *Management Science 36*, 2 (1987), 254–265.

[34] SAVELSBERGH, M. W. P. Local search in routing problems with time windows. *Annals of Operation Research 4* (1985/6), 285–305.

[35] SAXENA, S. *Design and Analysis of Some Combinatorial and Computational Geometry Problems for Parallel Execution.* PhD thesis, Comp. Sci. & Engg., IIT Delhi, January 1989.

[36] SOLOMON, M., AND DESROSIERS, J. Time window constrained routing and scheduling problems: A survey. *Transportation Science 22* (1988), 1–13.

[37] SPRINGSTEEL, F., AND STOJMENOVIC, I. Parallel general prefix computations with geometric, algebraic, and other applications. *International Journal of Parallel Programming 18*, 6 (1989), 485–503.

[38] SUNDER, S., AND HE, X. Scheduling interval ordered tasks in parallel. *Journal of Algorithms 26* (1998), 34–47.

[39] TAKAOKA, T. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters 43* (1992), 195–199.

[40] TAKAOKA, T. Sub-cubic cost algorithms for the all pairs shortest path problem. *Lecture Notes in Computer Science 1017* (1995), 333–343.

[41] TSITSIKLIS, J. N. Special cases of traveling salesman and repairman problems with time windows. *Networks 22* (1992), 263–282.

[42] ULLMAN, J. D. Complexity of sequencing problems. In *Computer and Job Scheduling Theory*, E. G. Coffman, Ed. 1976.

[43] VAZIRANI, U. V., AND VAZIRANI, V. V. The two processor scheduling is in random .NC. *SIAM Journal of computing 18* (1989), 1140–1148.